TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

BAKALÁŘSKÁ PRÁCE

Emulátor celulárních automatů na FPGA

Liberec 2013 Veronika Fulínová

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2646 – Informační technologie

Studijní obor: 1802R007 – Informační technologie

Emulátor celulárních automatů na FPGA

The emulator of the cellular automata on FPGA

Bakalářská práce

Autor práce: **Veronika Fulínová**

Vedoucí práce: Ing. Martin Rozkovec, Ph.D.

Konzultant práce: –

V Liberci 17. května 2013

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: 2012/2013

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Veronika Fulínová**
Osobní číslo:	**M09000132**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Emulátor celulárních automatů na FPGA**
Zadávající katedra:	**Ústav informačních technologií a elektroniky**

Zásady pro vypracování:

1. Seznamte se s teorií stavových automatů s důrazem na celulární automaty
2. Na PC naprogramujte simulátor celulárních automatů s uživatelsky volitelným počtem vazeb a konfigurovatelnou přechodovou funkcí
3. Navrhněte vhodnou strukturu hardwarového emulátoru
4. Experimentálně ověřte funkčnost emulátoru na vývojové desce s FPGA obvodem

Rozsah grafických prací:	**Dle potřeby dokumentace**
Rozsah pracovní zprávy:	**cca 30 stran**
Forma zpracování bakalářské práce:	**tištěná/elektronická**

Seznam odborné literatury:

[1] **Salcido A.: Cellular Automata - Simplicity Behind Complexity, InTech 2011, ISBN 978-953-307-230-2**

[2] **Wolfram S.: Cellular Automata and Complexity: Collected Papers, Westview Press 1994, ISBN 0-201-62716-7**

[3] **Počítačové architektúry a diagnostika, sborník semináře, STU v Bratislavě 2011, ISBN 978-80-227-3552-0**

[4] **Pinker J., Poupa M.: Číslicové systémy a jazyk VHDL, BEN 2006, ISBN 80-7300-198-5**

Vedoucí bakalářské práce:	**Ing. Martin Rozkovec** Ústav informačních technologií a elektroniky
Datum zadání bakalářské práce:	**1. října 2012**
Termín odevzdání bakalářské práce:	**20. května 2013**

prof. Ing. Václav Kopecký, CSc.
děkan

Technická univerzita v Liberci - 14 -

prof. Ing. Zdeněk Plíva, Ph.D.
pověřen vedením ústavu

V Liberci dne 1. října 2012

Prohlášení

Byla jsem seznámena s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/ 2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracovala samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum

Podpis

Poděkování

Na tomto místě bych ráda poděkovala především vedoucímu bakalářské práce Ing. Martinu Rozkovcovi, Ph.D. za cenné rady, pomoc a také za podporu při tvorbě této bakalářské práce.

Velké poděkování bych také ráda vyjádřila mé rodině a blízkým, kteří mě během vypracovávání práce ve všem podporovali.

Abstrakt

Bakalářská práce se zabývá studiem celulárních automatů a vývojem desktopové aplikace, která by jak simulovala jejich chování na základě uživatelem volených parametrů, tak umožnila generovat VHDL kód, díky kterému by bylo možné spustit výpočet následujících stavů celulárního automatu pomocí FPGA.

Klíčová slova: celulární automat, VHDL, FPGA

Abstract

Bachelor thesis deals with the study of the cellular automata and with the development of the desktop application that would simulate their behavior based on user-selectable parameters, allowing to generate VHDL code that makes it possible to run the calculation of the following states cellular automaton using FPGA.

Key words: cellular automaton, VHDL, FPGA

Obsah

Seznam obrázků

Úvod

Cílem bakalářské práce je seznámit se se základními vlastnostmi celulárních automatů a pochopit, na jakém principu mění každou iterací svůj stav. Úkolem je na základě získaných vědomostí ohledně této problematiky navrhnout vhodný algoritmus, jenž by dle určené přechodové funkce nastavoval následující hodnoty automatu a tím měnil jeho stav.

Výstupem této práce by měla být grafická desktopová aplikace, která by umožňovala jak simulovat chování jednodimensionálních (1D) a dvoudimensionálních (2D) celulárních automatů podle uživatelem volených pravidel, tak generovat VHDL zdrojový kód k emulaci automatů na FPGA.

Práce bude pomyslně dělena na dvě části – teoretickou a praktickou. Teoretická část bude zahrnovat vysvětlení základních pojmů, příklady možného využití celulárních automatů v praxi a popis vývojových programovacích jazyků a nástrojů. Do praktické části se bude řadit popis desktopové aplikace a kódu v jazyce VHDL, díky kterému je umožněno spustit výpočet následujících stavů automatu pomocí FPGA.

1. Základní pojmy a teorie

1.1. Stavový automat

Jak z názvu vyplývá, stavovým automatem se rozumí systém, který se využívá v případě, kdy se zkoumá či nastavuje chování nějakého objektu, který se může nacházet v různých stavech. Mezi těmito stavy se může samozřejmě přepínat (např. přecházet mezi stavem v log. 0 a log. 1) – a to na základě vnějších podnětů (vstupů) v diskrétním čase. Může být synchronní i asynchronní [18], [19].

1.1.1. Konečný automat

Nejznámějším stavovým automatem je konečný stavový automat. Jeho název se odvíjí od skutečnosti, že množina možných stavů automatu je konečná. Jelikož se jedná o automat patřící do rodiny stavových automatů, tak i zde se jedná o systém, který dle hodnoty vstupu mění svůj stav.

Model konečného stavového automatu nalézá mnohá uplatnění v praktickém užití. V podstatě jakýkoli předmět, který přijímá nějaké vstupní hodnoty, na jejichž základě vyhodnocuje svou reakci v diskrétním čase (počet možných reakcí je konečná množina stavů), je konečným stavovým automatem. Jedná se například o klasický automat na kávu, který vydává druh kávy dle zmáčknutého tlačítka.

Konečný automat může být deterministický (DKA) a nedeterministický (NKA). Deterministickým automatem se rozumí takový automat, který se vždy nalézá právě v jednom ze svých stavů. Nedeterministický může naopak být v celé množině stavů [9].

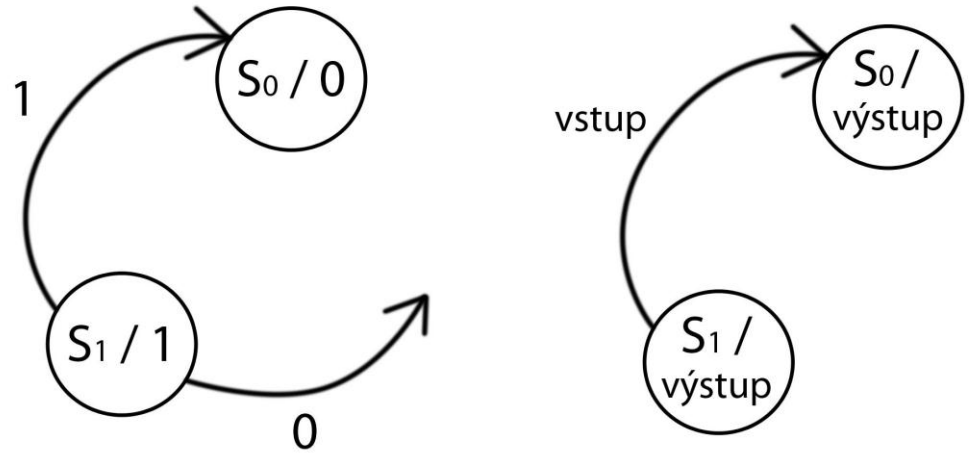
DKA je každá pětice prvků A = (Q, Σ, O, δ, λ), kde [17]:

- Q - konečná neprázdná množina stavů
- Σ - konečná neprázdná množina vstupních symbolů (vstupní abeceda)
- O - konečná neprázdná množina výstupních symbolů (výstupní abeceda)
- δ - přechodová funkce, δ: Q x Σ → Q
- λ - výstupní funkce, Q × Σ → O (Mealy); Q → O (Moore)

NKA je každá pětice prvků A = (Q, Σ, δ, I, F), kde [9]:

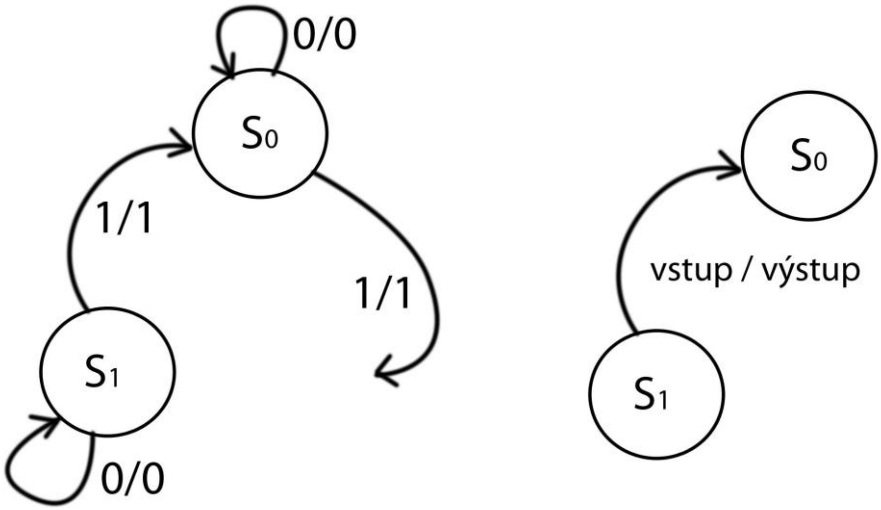
- Q - konečná neprázdná množina stavů
- Σ - konečná neprázdná množina vstupních symbolů (vstupní abeceda)
- δ - přechodová funkce, δ: Q x Σ → Q
- I - množina počátečních (iniciálních) stavů, I ⊆ Q
- F - množina koncových neboli přijímajících stavů, F ⊆ Q

Nejčastěji využívanými jsou automaty typu Mealy a Moore, které jsou navzájem převoditelné. Výstup automatu typu Moore je závislý pouze na aktuálním vnitřním stavu. Jak je vidět na uvedeném nákresu diagramu [Obr. 1], přechodové hrany mezi stavy jsou popsány pouze vstupní proměnnou a výstup je uveden přímo u každého ze stavů.



Obr. 1 - Mooreho automat

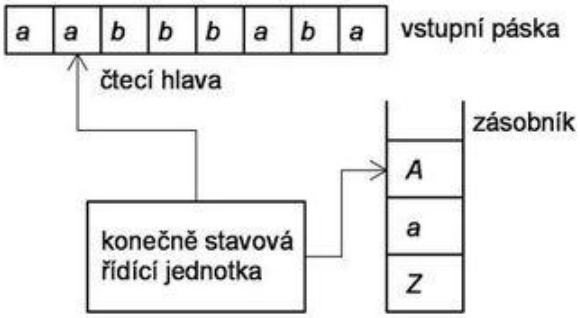
V případě automatu typu Mealy je však výstup závislý jak na současném vnitřním stavu, tak na aktuální hodnotě vstupu. Pro lepší představu je zde opět uveden diagram, na kterém jsou přechodové hrany popsány jak vstupní proměnnou (stejně jako u typu Moore), tak hodnotou výstupu – čísla jsou od sebe navzájem oddělena lomítkem [10].



Obr. 2 - Mealeho automat

1.1.2. Zásobníkový automat

Deterministický zásobníkový automat (DZA) je v podstatě konečný stavový automat rozšířen o zásobník, ve kterém jsou uloženy symboly [4]. Zásobník je typu paměti FIFO – ***F**irst **I**n **F**irst **O**ut* (první dovnitř – první ven). To znamená, že se přistupuje k hodnotám v takovém pořadí, v jakém byly nahrány do paměti a nahlíží se vždy jen na vrchol zásobníku, který může být pozměněn odebráním symbolu či přidáním nového nad něj – ten by tvořil nový vrchol. Na následujícím obrázku [Obr. 3] je velmi jednoduchá vizualizace toho, jak automat funguje.



Obr. 3 - Zásobníkový automat

Přidáním zásobníku se ovlivnil výpočet přechodové funkce, který je závislý na kombinaci hodnot aktuálního stavu, vstupní proměnné a vrcholu zásobníku (p, q_0, z_0). Písmeno $\mathbf{q_0}$ znamená počáteční stav a $\mathbf{z_0}$ značí vrchol zásobníku. Znakem **p** je označován vstup. Jelikož se ale v některých případech nemusí (na rozdíl od konečného stavového automatu) symbol ze vstupní pásky vůbec číst a je možné jej úplně vynechat, může být **p** nahrazeno tzv. prázdným znakem **λ**. Výpočet by pak probíhal z trojice (λ, q_0, z_0). Dle této funkce se udává, do jakého následujícího stavu má stroj přejít a jaká posloupnost znaků nahradí stávající vrchol.

DZA je každá sedmice $M = (Q, A, Z, \delta, q_0, z_0, F)$, kde [4]:

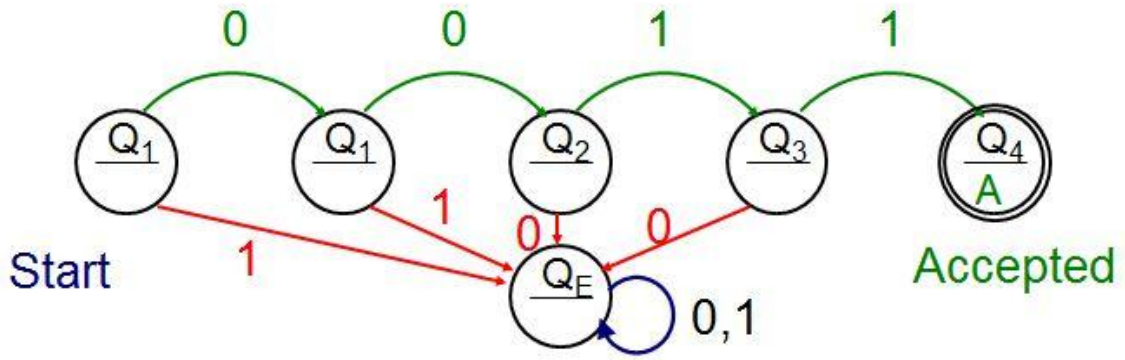
- Q - konečná množina stavů
- A - konečná vstupní abeceda (vstupní páska)
- Z - konečná zásobníková abeceda (zásobník)
- $\delta: Q \times (A \cup \{\lambda\}) \times Z \to Q \times Z^*$ - přechodová funkce
- $q_0 \in Q$ - počáteční stav,
- $z_0 \in Z$ - počáteční zásobníkový symbol (vrchol zásobníku)
- $F \subseteq Q$ - množina přijímacích stavů

1.1.3. Klasifikační automat

Jedná se o deterministický konečný automat, který má výstup jen v případě splnění požadované podmínky – jinak je bez výstupu. K tomuto účelu obsahuje stav,

který na závěr zhodnotí, zda podmínka byla či nebyla splněna - vygeneruje výstup A (Accepted = Přijato).

Na uvedeném obrázku značí zelené přechodové hrany stav, kdy je podmínka splněna. Na tomto konkrétním příkladu je vidět, že podmínka bude splněna, pokud počáteční kombinace vstupu bude odpovídat posloupnosti ‚0011'. Chybu na obrázku znázorňují červeně zbarvené hrany, které vedou do cyklu, který již nepřechází do žádného dalšího stavu a zacyklí se – nebyla totiž splněna podmínka, že zadané hodnoty budou touto kombinací začínat (platí pro tento příklad). Automat s tímto chováním je nazýván „synchronní kódový zámek".



Obr. 4 - Klasifikační automat

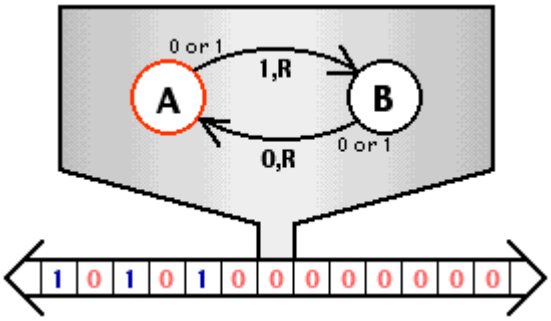
KKA je každá pětice A = (Q, Σ, δ, q_0, F), kde[17]:

- Q - konečná, neprázdná, množina stavů
- Σ - konečná neprázdná množina vstupních symbolů (vstupní abeceda)
- δ (přechodová funkce) - zobrazení $\delta: Q \times \Sigma \rightarrow Q$
  Přechod je určen stavem, ve kterém se automat nachází a symbolem, který přichází na vstup (nebo který je čten na vstupu)
- q_0 - počáteční stav ($q_0 \in Q$)
- F - množina koncových stavů ($F \subseteq Q$)

1.1.4. Turingův stroj

Turingův stroj (TS) je abstraktní model výpočetního zařízení, který se používá ke zjištění, zda se za jeho pomoci dají některé problémy vyřešit či nikoliv. Byl definován roku 1937 anglickým filozofem, matematikem a kryptografem Alanem Mathisonem Turingem.

Motivací k jeho vytvoření byl tzv. rozhodovací problém, který byl zkoumán matematikem Davidem Hilbertem na počátku 20. století. Řešilo se v něm, zda existuje algoritmus, který by uměl rozhodnout o pravdivosti libovolného matematického tvrzení. Za pomoci Turingova stroje bylo dokázáno, že takový postup stanovit nelze [11].



Obr. 5 - Turingův stroj

Na jednoduchém nákresu stroje je vidět, že se skládá z konečného stavového automatu s přechodovými funkcemi a nekonečného jednorozměrného pole prvků, který tvoří pomyslnou pásku (*angl. tape*). Pole je zpočátku naplněné prázdnými hodnotami. Vyobrazený zúžený pruh vedoucí k pásce symbolizuje čtecí a zapisovací hlavu (*angl. head*), která pracuje vždy právě s jednou buňkou (*angl. cell*). Tuto buňku hlava přečte a na základě přechodových pravidel přepíše odpovídající hodnotou. V dalším kroku se hlava posune o jednu buňku doleva či doprava a postup se opakuje.

Celý výpočetní proces tedy probíhá tak, že se již od prvního kroku cyklicky zjišťuje, jestli aktuální stav neodpovídá stavu koncovému. V takovém případě výpočet končí. Pokud se momentální stav liší od koncového, hlava přečte z pásky znak, nad kterým se zrovna nachází a dle přechodové funkce se změní vnitřní stav. Hlava poté přepíše zkoumaný znak a posune se o jeden krok doleva či doprava a celý postup se opakuje. Pokud ale v přechodové funkci není definován přechod pro aktuální stav a přečtený znak, výpočet se ukončí.

TS je šestice $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, kde [21]:

- Q - konečná množina stavů
- Γ - konečná množina páskových symbolů
- $\Sigma \subseteq \Gamma, \Sigma \neq \emptyset$ - konečná množina vstupních symbolů
- $\delta : (Q - F) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 0, +1\}$ - přechodová funkce
- $q_0 \in Q$ - počáteční stav
- $F \subseteq Q$ - množina koncových stavů

1.2. Celulární automat

Název celulární automat (označováno často zkratkou CA) vznikl dle anglického slova „cell“, což v překladu znamená „buňka“. Jedná se o diskrétní systém, jenž se zabývá samovolným chováním jednotlivých buněk [5] Ve většině případů se v CA

rozlišují pouze dvě barvy (tzv. binární automat). Černá barva značí „živou“ buňku a bílá naopak „mrtvou“. Existují ale i automaty, které užívají více barev.

Automat mění svůj stav na základě daného pravidla (lokální přechodové funkce), který si uživatel zvolí. Do samotného běhu již nikdo zasáhnout nemůže. Vstupní parametry se určují na začátku. Pak se pouze sleduje, jaký bude vývoj. Buňky mohou „umírat“, „ožívat“ či zůstávat ve stejném stavu.

Celulární automaty se vyskytují v různých podobách. Většinou jsou buňky zarovnány do pravidelné mřížky s políčky čtvercového tvaru. Mohou mít ale i tvar šestiúhelníku, kdy mřížka připomíná včelí plástve. Stejně tak se mohou vyskytovat v různých prostorových rozměrech, z nichž se nejčastěji vyskytují jednodimensionální a dvoudimensionální.

1.2.1. Historie

Prvním, kdo se začal zabývat problematikou celulárních automatů, byl maďarský matematik John von Neumann [Obr. 6]. Zajímala ho především sebe-reprodukce. Spolu se Stanislavem Ulamem [Obr. 7] se snažil vytvořit výpočetní systém, který by byl schopen vytvářet své vlastní kopie – tzv. se rozmnožovat. Své poznatky zveřejnil roku 1948 na přednášce týkající se obecné teorie automatů.

J. von Neumann si své poznatky zapisoval a ve formě cyklostylovaných fragmentů a poznámek z jeho přednášek se staly inspirací pro mnohé práce zaměřené na konstrukci automatů. Shrnutí jeho vědomostí a poznámek bylo vydáno post mortem pod názvem „Theory of Self-reproducing Automata“[1].

Zajímavé myšlenky čerpal také z prací vědců W. McCullocha, W. Pittse a A. Turinga. Dokázal spojit matematickou teorii s biologií a vytvořit automat, který simuloval vývoj živé hmoty (studoval mikroorganismy). Byl ale příliš komplikovaný. Mohl nabývat 29 různých stavů a obsahoval 200 000 buněk. Vytvořil pravidelnou mřížku buněk, přičemž každá buňka byla považována za samostatný automat, celá mřížka za organismus.

[1] http://www.joomla-gnu.com/tuxebooks/VonNeumann.pdf

Obr. 6 - John von Neumann

Obr. 7 - Stanislav Ulam

Obr. 8 - John Conway

S. Ulam přišel se zjednodušením – poradil Neumannovi elegantnější způsob řešení. Nastínil mu myšlenku celulárního automatu [14].

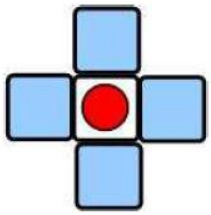
Prací J. von Neumanna a S. Ulama se nechal inspirovat britský matematik John Conway [Obr. 8], když našel pravidlo (lokální přechodovou funkci, platnou pro všechny buňky), které vedlo ke komplexnímu chování. V roce 1970 představil svůj automat jako „Hra života“ (*angl. Game of Life*), později proslavenou pod názvem „Conwayova hra života“ (*angl. Conway's Game of Life*) [7]. Té je věnována samostatná kapitola dále [1.3].

1.2.2. Rozdělení dle okolí

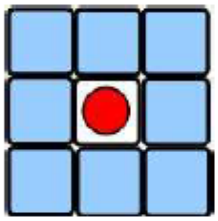
Každá buňka může nabývat různých stavů, nejčastěji jsou dvoustavové (tzv. binární CA), kdy mrtvá buňka je ve stavu log. 0 a živá v log. 1. Počáteční stav je dán pevně na vstupu a vývoj je dán přechodovými funkcemi (definované pravidly).

CA mohou být jednorozměrné (1D – pole), dvourozměrné (2D – mřížka), třírozměrné (3D) či více. Na základě uspořádání buněk se určuje vliv okolí na změnu stavu:

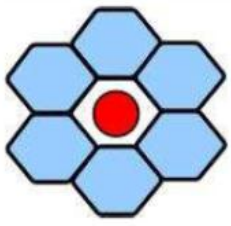
- 1D: buňka má dva sousedy
- 2D: existuje různé rozložení buněk určující počet sousedů
    - Von Neumannovo okolí (4 sousedé) [Obr. 9]
    - Moorovo okolí (8 sousedů) [Obr. 10]
    - Šestiúhelníkové okolí (6 sousedů) [Obr. 11]



Obr. 9 - Von Neumanovo



Obr. 10 - Moorovo



Obr. 11 - Šestiúhelníkové

1.2.3. Pravidla

Počet možných pravidel je dán tvarem CA (1D, 2D, atd.) a počtem sousedících a ovlivňujících buněk. V případě 1D uspořádání do pole má buňka 2 sousedy, následující její stav je tedy závislý na 3 buňkách – jí samotné a dvou sousedících. Její stav je popsán kombinací 2^3 (tj. 8) čísel (log. 0 a log. 1). Máme tedy 2^8 (tj. 512) definovaných pravidel. 2D CA s Moorovým okolím má 8 sousedů, takže stav prostřední ovlivňované buňky je dán kombinací 2^9 (tj. 512) binárních hodnot – celkově má 2^{512} možných pravidel [30].

1.3.Conwayova hra života

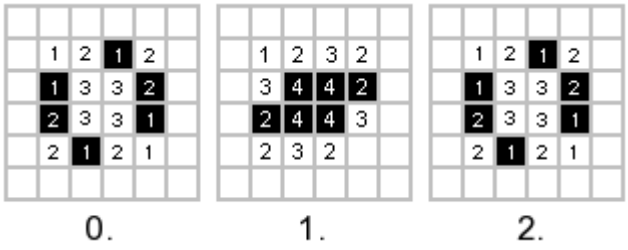
Conwayova hra života (*angl. Conway's Game of Life*) byla představena v roce 1970 proslulým britským matematikem Johnem H. Conwayem, dle kterého je nazývána [6]

Jedná se o konečný automat. Life nehraje žádný hráč. Výpočet je neinteraktivní, probíhá bez jakékoliv aktivní účasti uživatele – závisí pouze na vstupních parametrech.

Apriori je navržena pro dvourozměrné zobrazení. U jednorozměrných automatů dochází poměrně brzy k „záhynu" života, jelikož se tam nemůže použít celé pravidlo – jednotlivé prvky mají okolo sebe pouze dva sousedy.

Hra se řídí jednoznačně danými pravidly, podle kterých můžou nastat čtyři různé stavy – buňka „ožije", „umře", „přežije" či zůstane „mrtvá". Konkrétní znějí:

- OSAMĚNÍ – živá buňka umírá, má-li méně než 2 živé sousedy
- PŘELIDNĚNÍ – živá buňka umírá, má-li více než 3 živé sousedy
- NAROZENÍ – mrtvá buňka ožívá, má-li právě 3 živé sousedy
- PŘEŽITÍ – živá buňka zůstává žít, má-li 2 nebo 3 živé sousedy



Obr. 12 - Příklad vývoje ve třech generacích (vzor ropucha)

V každé iteraci se sleduje počet živých sousedů všech jednotlivých prvků a podle toho se nastaví jejich následující stav [Obr. 12], [13], [26].

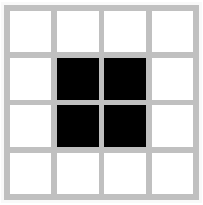
Jak již bylo výše zmíněno, hra je konečným automatem [1.1.1] o konečné množině stavů. Jaký stav bude buňka nabývat, se rozhoduje na základě přechodových funkcí určenými kritickými konstantami. Automat je deterministický – v každém kroku se nachází právě v jednom stavu. Množina koncových stavů je prázdná.

Ve hře se objevuje spousta možných kombinací chování buněk. Vzory se rozdělují do deseti základních kategorií, z nichž ty nejčastější jsou níže popsány a některé i zobrazeny.

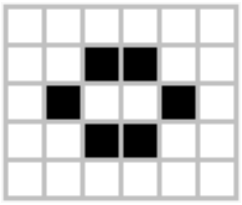
1.3.1. Still lives („Stále živí“)

Jedná se o stabilní neměnící se společenství buněk – svůj stav mohou změnit, pokud se k danému uskupení nepřiblíží jiné buňky. Mezi nejznámější z nich patří seskupení čtyř buněk situovaných do čtverce – tzv. bloky (*angl. blocks*). Dalšími častými vzory jsou úly (*angl. hives*), bochníky (*angl. loaves*) a loďky (*angl. boats*) [13].

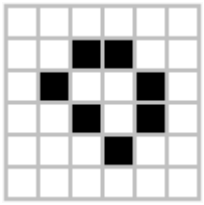
Poslední tři jmenované tvary se často vyskytují i ve větším seskupení a tvoří tak společenství úlů (tzv. medové farmy – *angl. honey farm*), bochníků (tzv. pekárny – *angl. bakery*) a lodí. V případě lodí nenese společenství žádné zvláštní označení [31].



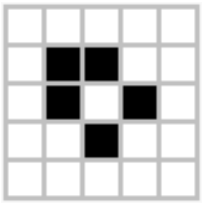
Obr. 13 - Blok



Obr. 14 - Úl



Obr. 15 - Bochník

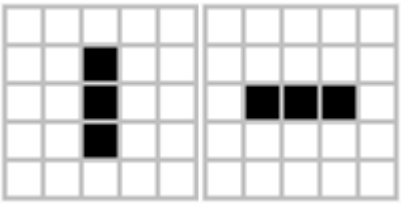


Obr. 16 - Loď

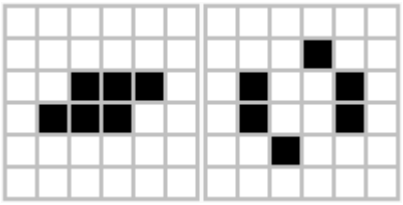
1.3.2. Oscillators („Oscilátory“)

Jedná se o vzor, který každou periodou cyklicky střídá svůj tvar. Nejčastěji se vyskytuje tzv. blikač, což jsou tři buňky v řadě a každým krokem se mění

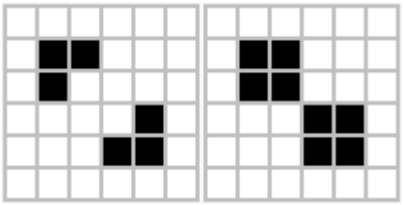
z vodorovného zobrazení na svislé a naopak. Mezi další ukázkové vzory se řadí ropucha, maják a pulzar (ten má periodu 3). Jelikož nelze na papír zobrazit pohyb, tak jsou jednotlivé kroky rozkresleny vedle sebe [13], [31].



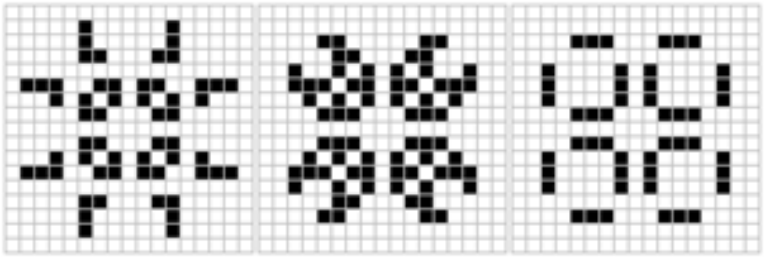
Obr. 17 - Blikač



Obr. 18 - Ropucha



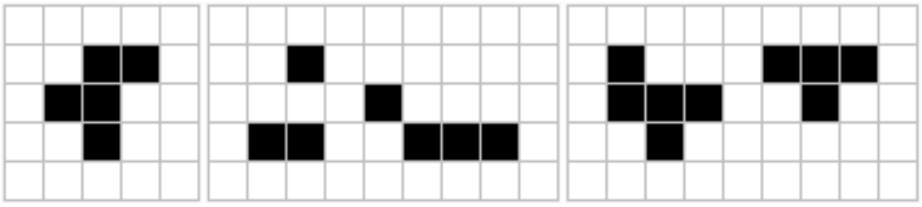
Obr. 19 - Maják



Obr. 20 - Pulzar

1.3.3. Methuselahs („Metuzalémové“)

Jakýkoliv malý vzor, jehož stabilizace trvá dlouhou dobu [Obr. 21], [13].

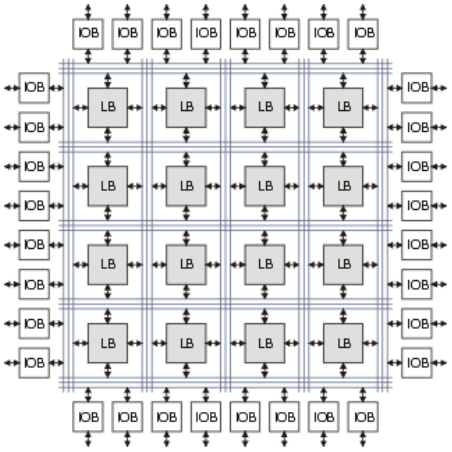


Obr. 21 - Metuzalémové

1.4.FPGA

Číslicové programovatelné obvody se souhrnně nazývají PLD (*angl. Programmable Logic Device*). Jde o obvody, u kterých si můžete naprogramovat jejich obsah a jejich vnitřní zapojení.

Číslicová programovatelná zařízení je možné podle způsobu programovatelnosti rozdělit do tří skupin. První skupinu tvoří klasické PLD, druhou CPLD (komplexní PLD) a do třetí skupiny patří obvody typu FPGA [8].

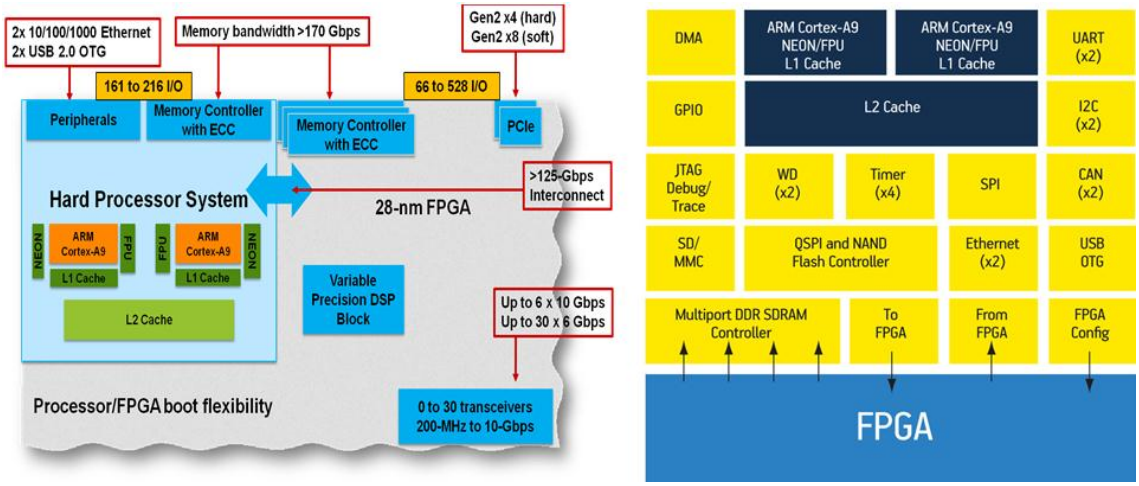


Obr. 22 - Vnitřní struktura FPGA (bloky)

Obvody FPGA (*angl. Field Programmable Gate Array*) jsou tvořeny seskupením několika bloků do matice. Na okrajích matice se nacházejí vstupně/výstupní bloky s označením IOB. Tyto bloky zpravidla obsahují registr, multiplexer, budič apod. Vnitřní strukturu matice tvoří programovatelné logické bloky (LB), které jsou mezi sebou navzájem různě propojeny [25].

1.5.AP SoC

AP SoC (All programmable System on Chip) je systém, který se skládá z jednoho a více procesorů, řadiče pamětí (externích a interních), sběrnicového propojení a specifických periferií – UART, VGA atd.



Obr. 23 - System on Chip © [1]

Jedná se v podstatě o FPGA na vyšší úrovni. Jak je z obrázků zřetelné, FPGA funkce je stále zachována – čip je pouze rozšířen o procesorový subsystém (procesory IBM, ARM, aj.).

Společnost Xilinx nabízí SoC pod názvem rodiny ZYNQ-7000. Na svých webových stránkách[2] vybízí uživatele k jejich využívání krátkým výčtem vlastností:

- nabízí inteligentnější systém
  - nejúčinnější ARM procesor + FPGA pro analýzu a řízení
  - nejrozsáhlejší OS
  - nejvyšší úroveň bezpečnosti a spolehlivosti
- bezkonkurenční provedení a výkon
  - dvoujádrové 1GHz ARM Cortex-A9 procesory
  - nejlepší provedení paměťového systému
  - nejnižší potřebné napájení a nejrychlejší logika
- osvědčená produktivita
  - špičkové syntézy na vysoké úrovni
  - nejširší sortiment softwarových prostředí a nástrojů

1.5.1. Zynq-7020

Konkrétním SoC, na kterém byl testován VHDL kód, je zařízení nesoucí název Zynq-7020 od společnosti Xilinx z rodiny Zynq-7000. V následující tabulce jsou vypsány hlavní parametry tohoto čipu [33].

Tab. 1 – Tabulka vlastností Zynq-7020

Processing System	**Procesor – jádro**	ARM Cortex Dual-A9 MPCore s CoreSight
	Procesor – rozšíření	NEON & Single/Double Precision Floating Point
	Maximální frekvence	667 MHz (-1); 733 MHz (-2); 800 MHz (-3)
	L1 cache	32 KB instrukce, 32 KB dat na procesor
	L2 cache	512 KB
	On-Chip memory	256 KB
	Podporované externí paměti	DDR3, DDR3L, DDR2, LPDDR2
	Podpora externích statických pamětí	2x Quad-SPI, NAND, NOR
	DMA kanály	8 (4 pro část „Programmable Logic“)

[2] http://www.xilinx.com/content/xilinx/en/products/silicon-devices/soc/zynq-7000.html

	Periférie	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO, 2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet (10/100/1000), 2x SD/SDIO
	Bezpečnost	RSA autentizace, AES a SHA 256b dešifrování a ověřování na Secure Boot
Propojení výpočetní části s programovatelnou logickou částí		2x AXI 32b Master 2x AXI 32b Slave
		4x AXI 64b/32b Memory
		AXI 64b ACP
		16 Interrupts

Programmable Logic	**Xilinx Série-7**	Artix-7 FPGA
	Programovatelné logické bloky	85 000
	Look-Up Tables (LUT)	53 200
	Flip-Flop klopné obvody	106 400
	RAM paměť	560 KB
	Programovatelné DSP	220
	Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs
	Bezpečnost	256b AES a SHA

2. Využití celulárních automatů

CA pro svůj nepředvídatelný vývoj mohou najít uplatnění ve všech vědních oborech, a to především v podobě simulátorů.

2.1. Příroda (biologie, geografie)

V celulárních automatech je nejdůležitější lokalita buňky a tvar jejího okolí. Proto jsou vhodným nástrojem pro studii jevů v přírodě, kde právě čas a prostor hraje důležitou roli. Praktické využití bychom mohli vidět na simulaci šíření požárů, možného dopadu kácení lesů v Amazonii nebo naopak v plánování zalesňování [16].

2.2. Sociologie (urbanizace)

Obor sociologie využívá celulární automaty k plánování rozvoje měst a studii urbanizace (proces koncentrace obyvatelstva do měst a s tím související změny kultury). Ke studiu růstu či úpadku města se využívají historické mapy, kde jsou pozorovány změny v určitých časových intervalech. Do analýzy vzniku měst se uvádějí úplné počátky, tj. od prvního záznamu, a sleduje se zde rozvoj periferních částí a centralizace či decentralizace obyvatel.

Zajímavým projektem bylo zkoumání růstu konkrétního města – např. Vídně, která zažila největší rozmach v období Vilémovského Německa (na poč. 20. st.) [16].

2.3. Návrhový nástroj

V současné době se CA používají také jako nástroje pro různé optimalizace, jako je projektování kanalizací, vedení podzemní vody, studie dopravního provozu (průchodnost křižovatek, ideálnost nastavení světelné signalizace) atd. [16]

2.4. Studie viru HIV

V posledních letech se věda zajímá o studii viru HIV a snaží se pochopit jeho chování a tím nalézt lék na uzdravení či zastavení šíření. I k tomu může dopomoci buněčný automat, který nám umožní při správné počáteční konfiguraci sledovat vývoj viru v krvi postiženého jedince a též zkoumat rozdíly u léčeného a neléčeného pacienta [16].

2.5.Studie fyzikálních jevů – teorie vzniku útvarů

Pomocí celulárních automatů se dají zkoumat různé fyzikální jevy, kde lze sledovat rozmach či úpadek. Krásným příkladem je studie vzniku sněhové vločky, která má téměř pokaždé jiný tvar. Další neméně zajímavou studií je zrození živého organismu či vznik vesmíru [16].

2.6.Generátor náhodných čísel

Jednou z vlastností celulárních automatů je jejich chování závislé na pravidlech a rozložení buněk – změní-li se nějak alespoň jedno z nich, bude vývoj automatu rozličný. Toho lze využít právě pro generování náhodných čísel a tím k šifrování [30].

3. Využité programovací jazyky a nástroje

3.1. Programovací jazyk Java

Java byla vyvinuta firmou Sun Microsystems a představena 23. května 1995 [15]. Java není výhradně určena pro specifickou aplikační oblast – lze v ní vytvářet jak desktopové aplikace, tak webové či server-client aplikace. Tím nabývá určité univerzálnosti.

Významným „soupeřem“ je jazyk C# od společnosti Microsoft. Dá se říct, že pokud umíte v Javě, neměl by být problém naučit se v C#.

3.1.1. Vlastnosti Javy

Tato část práce obsahuje výčet hlavních vlastností jazyka Java, díky kterým je programátory stále využíván.

Především je ryze objektově orientovaný, což ve zkratce znamená, že program tvoří hlavní třída a k ní přidružené vedlejší třídy. Běh programu je pak realizován jako využívání objektů a volání jejich metod. Zároveň se jedná o tzv. víceúlohový jazyk – podporuje užití vláken v programu. Je tzv. interpretovaný, což znamená, že místo skutečného strojového kódu se vytváří tzv. mezikód („byte-code“) [27]. Tento formát je nezávislý na architektuře zařízení. Program pak může tedy běžet na jakémkoli zařízení, který má k dispozici interpret Javy, tzv. virtuální stroj Javy (*angl. Java Virtual Machine – JVM*) [23]. Přestože se jedná o jazyk interpretovaný, není ztráta výkonu významná, neboť překladače pracují v režimu „just-in-time“ a do strojového kódu se překládá jen ten kód, který je opravdu zapotřebí.

Správa paměti je realizována pomocí tzv. garbage collectoru, který automaticky vyhledává již nepoužívané části paměti a uvolňuje je pro další použití – tím se liší od klasického jazyka C, kde si programátor musí paměti uvolňovat sám příkazem free().

Jazyk je také navržen pro podporu aplikací v síti (podporuje různé úrovně síťového spojení, práce se vzdálenými soubory, umožňuje vytvářet distribuované klientské aplikace a servery) [15].

3.2. Programovací jazyk C#

Jedná se o vysokoúrovňový objektově orientovaný jazyk vyvinutý firmou Microsoft, běžící na platformě .NET. Je založen na vlastnostech vícero jazyků, z čehož největší podíl tvoří Java a C/C++ (podobnost s jazykem Java je cca 80%).

První verze vyšla v roce 2002, kdy Bill Gates prohlásil, že na vývojářský trh přichází naprostá špička mezi ostatními. Byl tehdy představen spolu s celým vývojovým prostředím .NET [12].

K tomu, abyste mohli začít programovat v C# a využívali všechny jeho nabízené možnosti a funkce, potřebuje vývojář šikovný nástroj, jakým je například Microsoft Visual Studio – na tvorbu simulátoru byla využita verze 2010 [22].

Zmíněný software není zadarmo, ale můžeme využít třeba SharpDevelop, který je volně dostupný. Dalším hojně využívaným vývojovým prostředím je MonoDevelop, který je multiplatformní (avšak některé knihovny nefungují zcela správně či chybějí). Mono je projekt vedený firmou Novell (dříve firmou Ximian). Jeho cílem je vytvořit sadu nástrojů kompatibilních s prostředím .NET. K těmto nástrojům patří i překladač jazyka C# a Common Language Runtime. Mono může běžet na počítačích s operačními systémy Linux, FreeBSD, UNIX, Mac OS X, Solaris a Microsoft Windows.

Jak již bylo výše zmíněno, C# je spjatý s platformou .NET (či Mono). Chcete-li tedy spustit aplikace vytvořené v tomto jazyce, musíte mít na svém počítači nainstalován Microsoft .NET Framework, což je prostředí nutné pro běh .NET aplikací – jak spouštěcí rozhraní, tak potřebné knihovny. Pro majitele operačního systému Windows je k dispozici zdarma jako samostatná komponenta, která se do systému může doinstalovat.

Ale aby nevznikla mýlka – mezi jazyky pro vývoj .NET aplikací se řadí ještě např. F#, Visual Basic .NET, Delphi, J#, IronPython a Boo.

3.2.1. Vlastnosti C#

Jeho hlavní charakteristická vlastnost je uvedena již na začátku této podkapitoly – a to, že se jedná o ryze objektový jazyk. Dále:

- nabízí možnost využití vlastností a událostí
- správa paměti je automatická – o korektní uvolňování zdrojů aplikace se stará garbage collector

- podporuje zpracování chyb pomocí výjimek
- zajišťuje typovou bezpečnost [2]

3.3. Jazyk VHDL

Jazyk VHDL (***V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage*) je spolu s jazykem Verilog HDL jedním z nejpoužívanějších jazyků pro popis hardwarových struktur hradlových polí (CPLD, FPGA) nebo různých zákaznických obvodů (ASIC) [32]. Je určen pro návrh a testování elektronických systémů.

Základem „programování“ v jazyce VHDL jsou jeho bloky, na které se kód dělí. Je důležité poznamenat, že tímto jazykem se pouze popisuje chování hardwaru – nemluví se tedy o něm jako o programovacím jazyce. Zdrojový kód většinou obsahuje několik různě pospojovaných bloků, z čehož každý blok má své vstupy a výstupy a předem danou činnost [29].

Každý blok obsahuje dvě základné komponenty – entitu a architekturu. Entita definuje rozhraní, tedy vstupy a výstupy. Architektura popisuje její funkci [20]. VHDL podporuje také knihovny (standardní knihovna je STD) a různé package, které mohou obsahovat různé deklarace potřebných komponent či signálů a nemusí se tak deklarovat pokaždé zvlášť.

Příklad přidání knihovny:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

Příklad deklarace signálů:

```
signal A: std_logic := '1';
signal B: std_logic_vector(3 downto 0) := "2403";
signal C: std_logic_vector(3 downto 0) := (others => '1');
```

Mezi základní datové typy se řadí *std_logic*, který je využíván pro uložení jednoho znaku, a *std_logic_vector*, který je polem znaků.

3.3.1. Komponenta „Entity“

Tato komponenta definuje vstupní a výstupní signály daného modulu. Pokud je blok pouze součástí rozsáhlejšího návrhu, slouží jeho výstupní signály (porty) k propojení s nadřazeným blokem [28].

```
Syntaxe:
  ENTITY název_entity IS
  PORT(
    jméno_signálu1 : mód datový_typ;
    jméno_signálu2 : mód datový_typ
  );
  END název_entity;
```

Port

Porty popisují vstupy a výstupy Entity. Pro jejich nastavení je třeba zadat jejich název, mód a datový typ [28]. Módy známe:

- **IN** – data z portu lze pouze číst
- **OUT** – data vycházejí z portu
- **BUFFER** – obousměrný tok dat – aktivní může být vždy jen jeden směr
- **INOUT** – obousměrný tok dat – může být více aktivních signálů najednou
- **LINKAGE** – neznámý směr datového toku

```
Příklad:
  Port (
    a,b :  IN std_logic;
    sin :  IN std_logic_vector(7 downto 0);
    sout : OUT std_logic_vector(7 downto 0);
  );
```

Generic

V části *Generic* jsou deklarovány generické konstanty platné v dané instanci entity. Konstanty mohou být libovolného typu, pokud není typ definovaná generickou konstantou.

3.3.2. Komponenta „Architecture“

V této části kódu se popisuje chování Entity. Dělí se na dvě části – deklarační a příkazovou [28].

```
Syntaxe:
  ARCHITECTURE název_architektury OF název_entity IS
    signal název_signálu : datový_typ;
    variable název_proměnné : datový_typ;
    constant název_konstanty : datový_typ := hodnota;
  BEGIN
  -- popis funkce modulu
  END;
```

Architektura může být popsána různými styly jazyka VHDL. Nejčastěji se mluví o stylu behaviorálním, strukturálním a stylu popisujícím tok dat (tzv. data-flow).

3.3.3. Příkaz „Process“

Process je jedním z paralelních příkazů VHDL. Paralelními příkazy jsou: přiřazení, proces, generate, instance komponenty, procedura.

Procesů může být v kódu libovolný počet. Aktivní jsou pouze v okamžicích, kdy dojde ke změně proměnných uvedených v závorce v definici procesu. Při změně ostatních proměnných se proces neaktivuje.

Pokud jsou v těle procesu deklarovány nějaké proměnné, jsou viditelné a přístupné pouze v něm – stejně jako v případě metod v jiných jazycích.

4. Desktopová aplikace

Bakalářská práce se skládá ze tří částí, z čehož dvě jsou desktopové aplikace, které simulují běh celulárních automatů a třetí částí je program psaný v jazyce VHDL. Desktopové aplikace jsou psány v programovacích jazycích Java a C#, takže by se dalo říct, že se jedná o porovnání těchto dvou jazyků – jak se lišil jejich vývoj.

Simulátor původně vznikal v jazyce Java. Poté, co jsem se naučila pracovat s jazykem C#, učinila jsem rozhodnutí, že to zkusím napsat v něm.

4.1. Aplikace v jazyce Java

Program obsahuje dvě třídy, v kterých jsou popsány jednotlivá pravidla pro chování buňky:

- Pravidla1D.java – obecná pravidla celulárních automatů 1D
- Pravidla2D.java – obecná pravidla celulárních automatů 2D (2^{512})

4.1.1. Vstupní buňky

Základem pro běh automatu jsou vstupní buňky, které v každém kroku mění svůj stav na základě daných pravidel. Jelikož se po spuštění aplikace nedá do programu již nijak zasahovat, volí se všechny parametry hned na začátku.

Nejprve je zapotřebí zvolit, má-li se jednat o automat jednodimensionální (1D) či dvoudimensionální (2D) a zadat požadovaný rozměr. V případě 1D se jedná pouze o počet buněk na ose X. Jednotlivé kroky jsou tedy vypisovány pod sebe. Za to u 2D se zadává rozměr jak na ose X, tak na ose Y. Vznikne tak matice X × Y. Zde se již nevypisují změny pod sebe – matice se každým dalším krokem překresluje.

Poté, co je zvolen prostorový rozměr a velikost automatu, musejí se zadat hodnoty jednotlivých buněk – tedy 1 nebo 0. Aplikace nabízí tři možnosti vložení těchto hodnot. Nejjednodušší je tzv. random volba, kdy se na základě rozměru vygeneruje náhodný sled nul a jedniček. Ale pokud je požadavkem zkoumat chování nějakého konkrétního výchozího stavu, je nutné zvolit jednu z dalších dvou možností.

Jedná se o ruční zadání přímo v aplikaci, které ale při větším rozměru automatu není příliš přehledné. Pokud se ale pro tuto volbu rozhodnete, vyskočí na vás okénko, do kterého zapíšete odpovídající počet nul a jedniček oddělených čárkou. Poslední

možností je zadání ze souboru. Hodnoty vstupních buněk musejí být psány do jednoho řádku a odděleny pouze čárkou – bez mezer. Posledním znakem být čárka ovšem nesmí. Soubor musí být uložen ve formátu *.TXT.

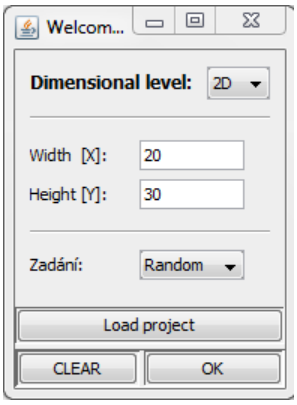
4.1.2. Pravidla

Po zvolení požadovaných vstupních buněk přichází na řadu volba pravidla, dle kterého se bude určovat následující stav. V první řadě je zapotřebí určit, zda mají mít všechny buňky jedno společné pravidlo (homogenní) či každá buňka jiné (heterogenní).

Protože je tedy výsledná desktopová aplikace psána v jazyce druhém, podrobnější informace o pravidlech naleznete v následující kapitole [4.2.2].

4.1.3. GUI – uživatelské rozhraní a ovládání hry

Po spuštění aplikace se zobrazí okénko, kde bude možnost zvolit prostorový rozměr (1D či 2D) a počet buněk na každé z aktivních os. Po nastavení těchto základních parametrů je třeba určit, jakým způsobem budou do aplikace nahrány hodnoty jednotlivých buněk [4.1.1]. Pokud však existuje nějaký uložený projekt, je možné jej kompletně nahrát stisknutím tlačítka „Load project“. Poté se automaticky přejde do hlavního okna, odkud je možné automat spustit.



Obr. 24 - Vstupní buňky

Obr. 25 - Pravidla automatu

Jinak se pokračuje do sekce nastavení pravidel, kde se volí mezi homogenním a heterogenním. Posléze se nastavují hodnoty pravidel. U jednorozměrného celulárního automatu je nejvyšším možným pravidlem hodnota 2^8 (256). U dvourozměrného automatu je to číslo poněkud vyšší – 2^{512}. Proto se pro zápis využívá hexadecimální soustava.

Po kompletním nastavení vstupních parametrů se zobrazí hlavní okno aplikace, ve kterém lze spustit simulace celulárního automatu, která se může pomocí komponenty *slider* zrychlovat či zpomalovat.

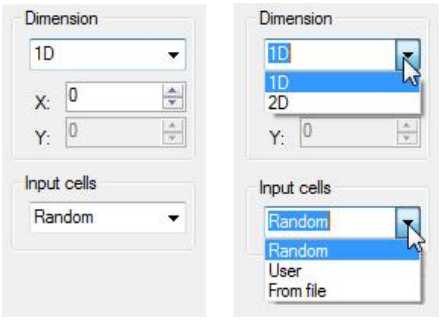
4.2. Aplikace v jazyce C#

I tato aplikace využívá objektově orientované programování. Program obsahuje čtyři třídy, v kterých jsou popsány jednotlivá pravidla pro chování buňky:

- ConwaysGameOfLife_1D.cs – pravidlo Conwayovy hry života 1D
- ConwaysGameOfLife_2D.cs – pravidlo Conwayovy hry života 2D
- Rules_1D.cs – obecná pravidla celulárních automatů 1D
- Rules_2D.cs – spirální pravidla celulárních automatů 2D
- Rules_2D_hexa.cs – obecná pravidla celulárních automatů 2D (2^{512})

4.2.1. Vstupní buňky

Jak již bylo zmíněno u Java aplikace, základním kamenem pro celulární automat jsou počáteční vstupní buňky, kde záleží jak na jejich počtu, tak na hodnotách každé z nich.



Obr. 26 - Volby možností vstupních buněk CA

I tato aplikace nabízí možnost volby mezi jednodimensionálním či dvoudimensionálním prostorovým rozměrem, kdy se poté udává počet prvků na ose/osách. Tím jsou tedy udány prvotní parametry tvaru automatu, který pak zbývá jen naplnit hodnotami buněk (opět jedničky a nuly) a to rovněž třemi možnými způsoby – ručně, ze souboru či si je nechat náhodně vygenerovat [Obr. 26].

Velikost automatu je omezena na rozměr 60 × 50 vstupních buněk.

4.2.2. Pravidla

Druhým důležitým faktorem ovlivňujícím chování celulárního automatu jsou pravidla, dle kterých se mění následující stav buněk. Základní rozdělení je na heterogenní, kdy má každá buňka své vlastní pravidlo, a homogenní, kdy je jedno pravidlo společné pro všechny buňky. I v heterogenním případě může nastat, že většina ze vstupních buněk může mít stejné pravidlo. V této bakalářské práci je zahrnuto ještě navíc jedno specifické pravidlo – Conwayova hra života (*angl. Conway's Game of Life*), které bylo popsáno již výše [1.3].

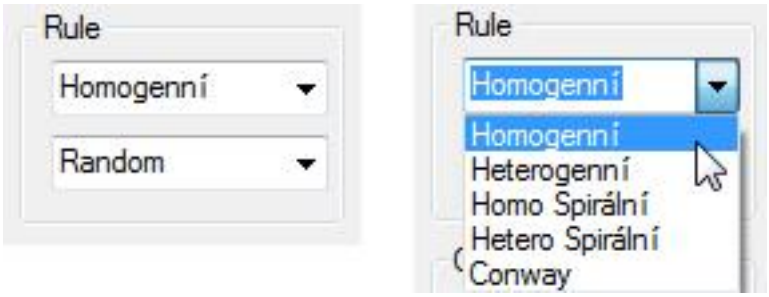
```
//Nastavení následujícího stavu dle pravidel Conwayovy hry života
public int[,] setState() {
int pomI_1, pomI_2, pomJ_1, pomJ_2; //pomocné při okrajových podmínkách
int i = 0, j = 0, k = 0, l = 0;     //indexy
int pocetBunek = this.x * this.y;   //počet procházení polem
int countLive = 0;                  //počet živých prvků
krok = new int[this.y,this.x];      //zápisné pole

while (pocetBunek > 0) {
        countLive = 0;
        //okrajove podminky - cyklicke reseni
        if ((i - 1) < 0) pomI_1 = this.y;               else pomI_1 = i;
        if ((i + 1) >= this.y) pomI_2 = -1;             else pomI_2 = i;
        if ((j - 1) < 0) pomJ_1 = this.x;               else pomJ_1 = j;
        if ((j + 1) >= this.x) pomJ_2 = -1;             else pomJ_2 = j;

        //nastaveni aktualniho prvku a jeho okoli
        this.aktual[0] = inputField[pomI_1 - 1, pomJ_1 - 1];
        this.aktual[1] = inputField[pomI_1 - 1, j];
        this.aktual[2] = inputField[pomI_1 - 1, pomJ_2 + 1];
        this.aktual[3] = inputField[i, pomJ_1 - 1];
        this.aktual[4] = inputField[i, j];  //aktualni prvek
        this.aktual[5] = inputField[i, pomJ_2 + 1];
        this.aktual[6] = inputField[pomI_2 + 1, pomJ_1 - 1];
        this.aktual[7] = inputField[pomI_2 + 1, j];
        this.aktual[8] = inputField[pomI_2 + 1, pomJ_2 + 1];

        for (int q = 0; q < 9; q++){ (this.aktual[q] == 1) countLive++;}
        if (this.aktual[4] == 1)
        {
                if ((countLive == 3) || (countLive == 4)) {krok[k, l] = 1; }
                else if ((countLive < 3) || (countLive > 4)) {krok[k, l] = 0;}
        }
        else if (this.aktual[4] == 0) {
                if (countLive == 3) krok[k, l] = 1; else krok[k, l] = 0;
        }
        if (j < (x - 1)) { j++;} else {i++;j = 0;}
        if (l < (x - 1)) {l++;} else {k++;l = 0;}
        pocetBunek--;
}
return krok;
}
```

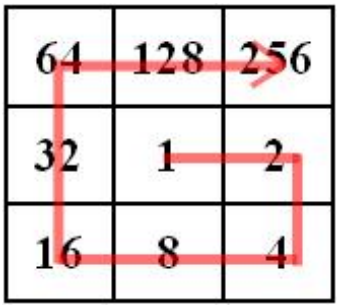
Pokud si uživatel zvolí Conwayovu hru života, už se samozřejmě nemusí starat o další zadávání hodnoty pravidla. V případě volby homogenního nebo heterogenního pravidla je opět nabídnuta možnost nahrání hodnot ze souboru, ruční zadání či si je nechat náhodně vygenerovat (obdobně jako u vstupních buněk [4.1.1]).



Obr. 27 - Nastavení pravidel

Počet pravidel pro 1D je 2^8 (tj. 256) a pro 2D je 2^{512}, což je již velmi vysoké číslo a už samotný zápis pravidla o 512 bitech je v decimálním tvaru nemožný. Proto se to dá řešit též pomocí tzv. spirálového pravidla [24].

Jak je vidět na obrázku [Obr. 28], každá buňka má přiřazené pravidlo. Jsou to mocniny dvou psané spirálně od středu matice po směru hodinových ručiček. Dle zadaného pravidla se určí, které sousední buňky budou určovat následující stav aktuálně zkoumané (středové).



Obr. 28 - Spirální pravidlo

Spirální způsob nastavování následujícího stavu je založen především na logickém součtu všech buněk, které byly na základě zvoleného pravidla určeny. Jelikož tento součet je nulový, pokud se sčítá sudý počet jedniček a naopak je roven jedné, sčítá-li se počet lichý, byl použit datový typ *bool*. Ten lze díky svým vlastnostem jednoduše měnit dle počtu nalezených živých sousedů.

Pokud bude zadáno například 183, převede se z decimálního tvaru na binární (0 1011 0111 → 128 + 32 + 16 + 4 + 2 + 1 = 183). Znamená to tedy, že stav buňky budou ovlivňovat hodnoty následujících prvků:

64	128	256
32	1	2
16	8	4

Stav středové buňky se tedy určí na základě logického součtu hodnot všech těchto buněk. Pro názorný příklad uvedu část matice, kde okrajovým buňkám přikládám hodnotu 0 na rozdíl od řešení v aplikaci – tam se problém s okrajem řeší cyklicky.

1	1	0	1
1	0	1	0
0	1	1	0
1	0	0	0

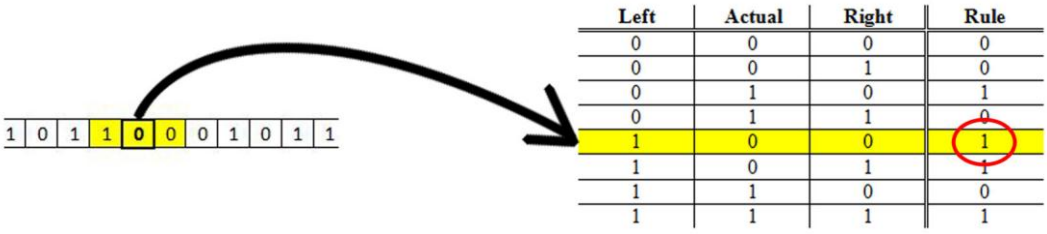
⇒

0	0	1	0
1	0	0	1
0	1	1	1
1	0	1	0

V matici vlevo je zadaný vstup a vpravo je výstup po jednom kroku na základě pravidla 183.

Spirální způsob nastavování následujícího stavu buněk je však implementován pouze do desktopové aplikace. Do VHDL kódu pro emulaci chování celulárních automatů na FPGA je generováno pouze klasické určování dle celého rozsahu 512 bitového pravidla. To je samozřejmě zavedeno i do desktopové aplikace a tím je umožněno využití celého prostoru 2^{512} možných pravidel. Vzhledem k velikosti zadávané hodnoty pravidla je užit zápis v hexadecimální formě o 128 znacích, což je sice velké číslo, ale zápis je praktičtější než ve formě decimální.

Následující stav se určuje dle aktuálního prvku a jeho okolí. Vhodným seřazením do vektoru se získá adresa, kde je uložena výsledná hodnota.



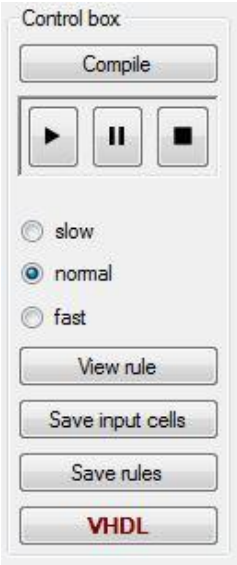
Obr. 29 - Adresa následující hodnoty buňky

Na tomto obrázku [Obr. 29] je vyobrazeno, jakým způsobem se získává následující stav u jednorozměrného automatu. V tomto konkrétním případě bude mít buňka v následujícím kroku hodnotu ‘1‘.

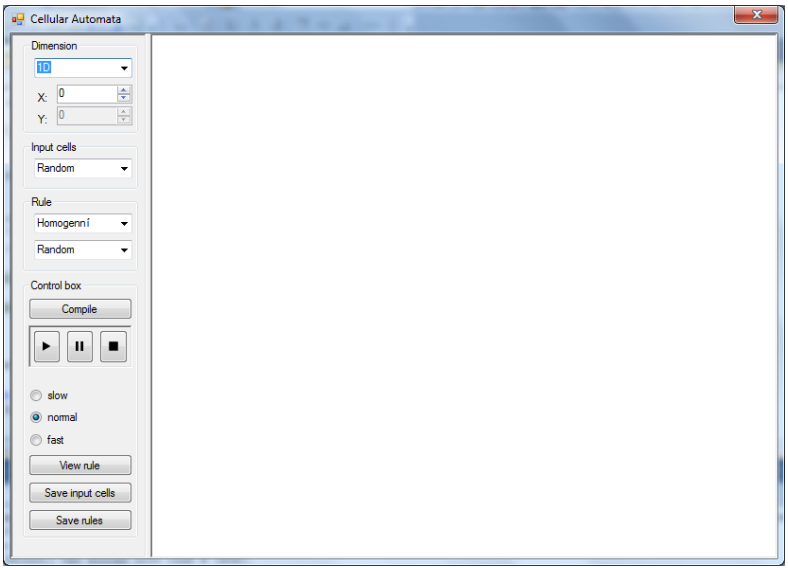
4.2.3. GUI – uživatelské rozhraní a ovládání automatu

Po spuštění programu se zobrazí okno rozdělené na dvě hlavní části – ovládací panel [Obr. 30] a prázdná plocha [Obr. 31] pro vykreslení. Panel pro obsluhu CA je dále členěn do čtyř částí, aby bylo zadávání vstupních parametrů pro uživatele co nejjednodušší. V první opticky oddělené části je umístěno rozhraní pro volbu mezi jednodimensionálním či dvoudimensionálním CA a numerické editory k nastavení jednotlivých hodnot v osách rozměru. V dalším kroku má hráč nabídku ze tří možností nahrání vstupních buněk – náhodně vygenerované, zadané ze souboru či ručně v textovém editoru programu. Poté následuje již poslední část, která výrazně ovlivňuje chování buněk. Jedná se o pravidla. Ty se opět dělí – na homogenní (jedno pravidlo společné pro celý CA) a heterogenní (každý prvek má své pravidlo). Bonusovou složkou v nabídce je Conwayova hra života.

Po zadání vstupních hodnot už zbývá pouze stisknout tlačítko „Compile“ a spustit tlačítkem „Play“. Hru můžete pozastavovat nebo úplně zastavit tlačítkem „Stop“ (smaže se obrazovka a mohou se nastavit jiné parametry). Buňky se pohybují v předem nastavené rychlosti, která byla určena za normální. Chod hry se pomocí tlačítek (tzv. radio button) mění na pomalejší či rychlejší.



Obr. 30 - Ovládací panel



Obr. 31 - Okno aplikace

Pro přehlednost bylo přidáno tlačítko, umožňující zobrazení zvolených pravidel. Bude-li se jednat o homogenní pravidlo, vypíše se samozřejmě pouze jediné. Půjde-li ovšem o heterogenní, vypíší se všechna pravidla (oddělená čárkou) vedle sebe – dle pořadí buněk, kterým naleží.

Po spuštění programu se do prázdného panelu začnou vypisovat čtverečky různých barev dle jejich hodnot (1 – černá; 0 – bílá), které se v případě jednorozměrného celulárního automatu postupně vykreslují pod sebe – co řádek, to nová generace buněk. U 2D automatu se čtverce s každým krokem překreslují [Obr. 38].

Pokud by uživatel měl zájem vidět ještě někdy přesně tu samou simulaci, může si vstupní buňky i pravidla uložit do textového souboru a později využít. Je ale zapotřebí pamatovat si rozměr automatu, proto by bylo vhodné poznamenat si tuto velikost například do názvu souboru. To je samozřejmě jen doporučení.

```
//Vykreslení řádku 1D CA
Rectangle r;
Brush black = Brushes.Black;
Brush white = Brushes.White;
Graphics g = panel2.CreateGraphics();

for (int i = 0; i < widthX; i++) {      //vykreslení jednoho řádku prvků
        bit = inputCells_1D[i];
        r = new Rectangle(actualX, actualY, rectSize, rectSize); //nastavení parametrů čtverce
        if (bit == 0) g.FillRectangle(white, r);        //vykreslení mrtvé buňky
        else g.FillRectangle(black, r);                 //vykreslení živé buňky
        actualX += rectOffset;          //nastavení mezery mezi jednotlivými buňkami
}

//Vykreslení matice 2D CA
for (int i = 0; i < heightY; i++) {
        for (int j = 0; j < widthX; j++) {
                bit = inputCells_2D[i,j];
                r = new Rectangle(actualX, actualY, rectSize, rectSize); //parametry čtverce
        if (bit == 1) g.FillRectangle(black, r);        //vykreslení živé buňky
                else g.FillRectangle(white, r);                 //vykreslení mrtvé buňky
                actualX += rectOffset;  //nastavení mezery mezi prvky v ose X
        }
        actualY += rectOffset;          //nastavení mezery mezi prvky v ose Y
        actualX = 0;
}
actualX = 0;
actualY = 0;
```

Uvedený úryvek kódu popisuje grafické vykreslení obou variant automatů, tj. jednodimensionálního do řádku a dvoudimensionálního do matice.

Posledním, avšak pro tuto bakalářskou práci nejdůležitějším, bodem je tlačítko „VHDL“, které umožňuje samotnou emulaci celulárních automatů na FPGA generováním zdrojových kódů v jazyce VHDL.

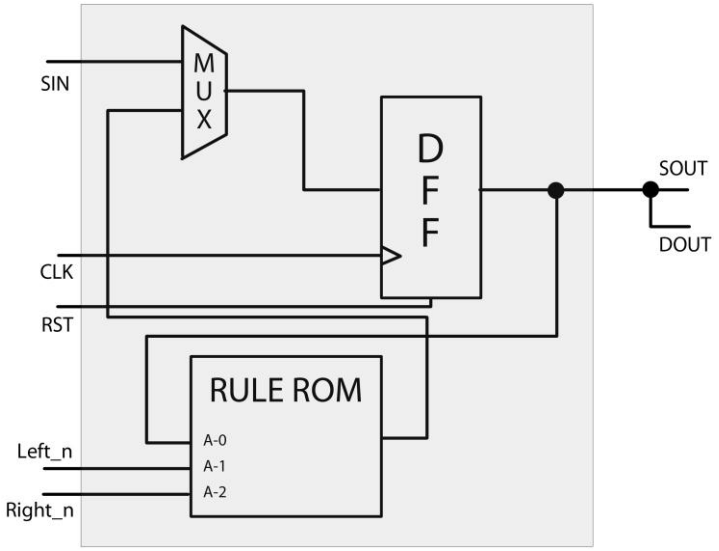
5. Celulární automat v jazyce VHDL

Jak již z názvu vyplývá, hlavním cílem bylo vytvořit takový program, pomocí kterého by se chování celulárních automatů vypočítávalo na FPGA. K tomuto účelu bylo na desktopové aplikaci umístěno tlačítko s názvem „VHDL“. Po řádném nastavení vstupních parametrů se musí celý automat zkompilovat. Poté je připraven jak pro spuštění simulátoru na panelu aplikace, tak pro vygenerování odpovídajícího VHDL kódu stisknutím výše zmíněného tlačítka.

Generuje se kód přesně odpovídající zadání. To znamená, že je do zdrojového kódu zanesen prostorový rozměr (1D či 2D), počet buněk na každé z využitých os a pravidlo platné pro každou buňku.

5.1. Jednorozměrný automat

Základním kamenem celulárního automatu je buňka (popsána v bloku CAC_1D_g.vhd), jejíž následující stav je ovlivněn jejím okolím a přechodovou funkcí. V případě jednorozměrného pole má zkoumaná buňka dva sousedy – levého a pravého. To vše je zahrnuto do hlavní komponenty VHDL kódu – entity CAC_1D_g.



Obr. 32 - 1D buňka VHDL

Na výše uvedeném obrázku je jednoduchý nákres jednorozměrné buňky celulárního automatu, jehož podrobnější popis bude následovat.

```
entity CAC_1D_g is
  Generic(
    rule_ROM : std_logic_vector(7 downto 0) := conv_std_logic_vector(90, 8)
  );
  Port(
    clk             : in  STD_LOGIC; -- hodiny
    clk_en          : in  STD_LOGIC; -- zastaveni vypoctu
    rst             : in  STD_LOGIC; -- reset automatu
    left_n          : in  STD_LOGIC; -- levy soused
    right_n         : in  STD_LOGIC; -- pravy soused
    shift_not_count : in  STD_LOGIC; -- nastaveni skenu automatu
    sin             : in  STD_LOGIC; -- seriovy vstup
    sout            : out STD_LOGIC; -- seriovy vystup == dout
    dout            : out STD_LOGIC  -- datovy vystup == sout
  );
end CAC_1D_g;
```

Jak je z tohoto úryvku kódu zřetelné, entita zahrnuje aktuálně zkoumaný prvek v podobě sériového vstupu SIN a dva sousední prvky LEFT_N a RIGHT_N. Výpočet je řízen hodinovým vstupem CLK, jenž je povolován signálem CLK_EN. Výsledek je pak poslán do signálů SOUT a DOUT.

V popisu architektury, tedy chování dané entity, jsou deklarovány ještě doplňující signály, týkající se pravidel a výsledné hodnoty buňky. Aktivní část architektury obsahuje dva procesy – skenovací scan_cell_mux_inst a resetovací či nastavovací CA_reg_DFFCE_inst.

V prvním z nich se na základě hodnoty signálu shift_not_count nastavuje hodnota proměnné CA_input. Pokud je shift_not_count rovno jedné, je do CA_input přivedena hodnota vstupního prvku SIN. V opačném případě je CA_input nastaveno na hodnotu vypočtenou na základě určeného pravidla.

```
rule_ROM_address <= left_n & CA_reg & right_n;
rule_ROM_result  <= rule_ROM(CONV_INTEGER(rule_ROM_address));
```

Aby mohla být výsledná hodnota vypočtena, je zapotřebí podívat se, jaká hodnota se nachází na adrese pravidla, která je určena třemi prvky – zkoumanou buňkou a jejími sousedy. Hodnota signálu rule_ROM_result je v podstatě výsledek.

V druhém procesu je pak řešeno to, zda bude výsledná hodnota CA_input přidělena do paměťové pomocné buňky CA_reg či bude tato buňka nulována (resetována). Hodnota této proměnné je poté nahrána na výstup.

Tento blok popisující základní chování buňky zůstává stále stejný. Měněn je blok druhý (CA_g.vhd), z kterého je zřetelné, jakou velikost automat má a jaké jsou jednotlivým buňkám přidělena pravidla.

```
entity CA_g is
  generic(
    CELL_NUM       : integer   := 256;  CIRCULAR       : integer   := 1;
    BOUNDARY_LEFT  : std_logic := '0';  BOUNDARY_RIGHT : std_logic := '0'
  );
  port(
    clk             : in  std_logic;  clk_en          : in  std_logic;
    rst             : in  std_logic;  shift_not_count : in  std_logic;
    sin             : in  std_logic;  sout            : out std_logic  );
end CA_g;
```

V generické části se nastavuje zmíněná velikost automatu (počet buněk) a způsob, jakým se budou řešit okrajové buňky. Pole prvků tvoří pomyslnou jednorozměrnou pásku konečné velikosti. Je-li hodnota proměnné CIRCULAR rovna jedné, je pole tzv. zacyklené. To pro představu znamená, že se konce pásky spojí. Takže levý soused prvního prvku je tvořen posledním prvkem pole.

Pokud však bude CIRCULAR nabývat nulové hodnoty, nebude páska spojena do kruhu. Místo toho budou krajové prvky pevně stanovena – levý soused první buňky a pravý soused poslední buňky bude roven hodnotě nula. Tato možnost ovšem není do desktopové aplikace implementována, a tak bude vždy generován kód se zacyklením.

```
-- zde se vytváří zacyklení posledního prvku na první
circular_CA : if CIRCULAR = 1 generate
  cell_connect(0)            <= cell_connect(CELL_NUM);
  cell_connect(CELL_NUM + 1) <= cell_connect(1);
end generate circular_CA;

-- zde se vytváří automat bez zacyklení – nulové okraje
noncircular_CA : if CIRCULAR = 0 generate
  cell_connect(0)            <= BOUNDARY_LEFT;
  cell_connect(CELL_NUM + 1) <= BOUNDARY_RIGHT;
end generate noncircular_CA;
```

Tento blok obsahuje pole pravidel, která byla generována v desktopové aplikaci dle určení uživatele. Toto pole o velikosti počtu prvků obsahuje vektory o velikosti 8 prvků (bitů) – pravidlo pro každou buňku.

Podle prvního bloku se zde pomocí FOR cyklu generuje jednorozměrný celulární automat dle počtu prvků. Zde se již při určování sousedních prvků počítá s variantou zacyklení pole a propojení do skenovacího řetězce.

```
CA_Scheme : for i in 0 to CELL_NUM - 1 generate
  CAC_1D_g_inst : entity work.CAC_1D_g
    generic map(
      rule_ROM => ROMs(i)
    )
  port map(
    clk             => clk,
    clk_en          => clk_en,
    rst             => rst,
    left_n          => cell_connect(i - 1 + 1),
    right_n         => cell_connect(i + 1 + 1),
    shift_not_count => shift_not_count,
    sin             => scan_connect(i - 1 + 1),
    sout            => scan_connect(i + 1),
    dout            => cell_connect(i + 1)
  );
end generate CA_Scheme;
```

5.2. Dvourozměrný automat

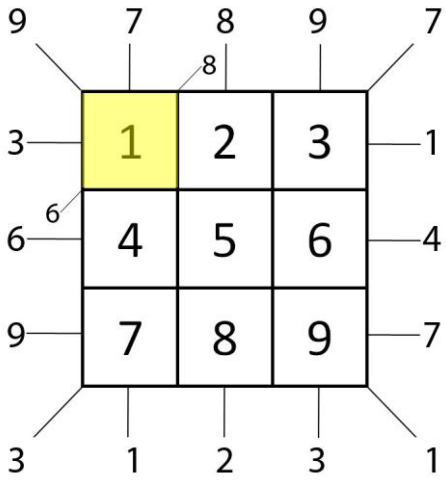
V případě dvourozměrného automatu jsou generovány opět dva bloky VHDL kódu – jeden obsahující popis chování jednotlivé buňky a druhý s konkrétním nastavením automatu.

Dvoudimensionální automat je zobrazován jako 2D mřížka o konečné velikosti. Proto má kolem sebe zkoumaná buňka více než dva sousedy. Jelikož se v této bakalářské práci pracuje s Moorovským okolím buňky, je počet sousedů roven číslu osm.

```
entity CAC_2D_g is
Generic(
GENERATE_SCAN : integer                    := 0;
rule_ROM  : std_logic_vector(511 downto 0) := conv_std_logic_vector(91, 512)
);
  Port(
    clk             : in  STD_LOGIC; -- hodiny
    clk_en          : in  STD_LOGIC; -- zastaveni vypoctu
    rst             : in  STD_LOGIC; -- reset automatu
    up_left_n       : in  STD_LOGIC; -- levy horni soused
    up_center_n     : in  STD_LOGIC; -- prostredni horni soused
    up_right_n      : in  STD_LOGIC; -- pravy horni soused
    mid_left_n      : in  STD_LOGIC; -- levy prostredni soused
    mid_right_n     : in  STD_LOGIC; -- pravy stredni soused
    down_left_n     : in  STD_LOGIC; -- levy dolni soused
    down_center_n   : in  STD_LOGIC; -- stredni dolni soused
    down_right_n    : in  STD_LOGIC; -- pravy stredni soused
    shift_not_count : in  STD_LOGIC; -- nastaveni skenu automatu
    pin             : in  STD_LOGIC; -- preset vstup
    sin             : in  STD_LOGIC; -- seriovy vstup / preset
    sout            : out STD_LOGIC; -- seriovy vystup == dout
    dout            : out STD_LOGIC -- datovy vystup == sout
  ); end CAC_2D_g
```

V ukázce kódu je popsána základní komponenta 2D buňky. Je ve většině shodná s 1D buňkou. Největším rozdílem je zmíněný počet sousedních buněk. Zbytek bloku je téměř shodný s popisem architektury 1D buňky [5.1].

Jak již bylo zmíněno výše, konkrétní nastavení velikosti automatu a jednotlivých pravidel se nachází v bloku CA2D_g.vhd. Stejně jako u jednodimensionálního automatu i zde jsou dvě možnosti, jak řešit sousedy okrajových buněk.



Obr. 33 - Zacyklené okrajové podmínky

Z desktopové aplikace bude mřížka vždy zacyklena – tvořit pomyslnou kouli. Na obrázku je jednoduchá demonstrace cyklického okolí buňky.

Architektura druhého bloku dvourozměrného automatu se od jednorozměrného liší v možnosti nastavení Conwayovy hry života, kdy se stav prvku mění v závislosti na počtu „živých“ sousedů v jakémkoliv rozmístění kolem buňky.

Uživatelem volená pravidla jsou zde stejně jakou u 1D automatu generována do dvourozměrného pole o velikosti počtu buněk a vektoru o rozměru 512 prvků (velikost pravidla). Pravidla jsou psána v hexadecimálním formátu. Využitím implicitní vlastnosti jazyka VHDL lze jednoduše převádět hodnoty z hexadecimálního zápisu na binární – umístěním znaku ’X‘ před řetězec znaků v šestnáctkové soustavě uzavřený do uvozovek.

Provedením syntézy náhodného dvoudimensionálního automatu byly získány orientační údaje o náročnosti výpočtu. Bylo zjištěno, že maximální možná frekvence se pohybuje v hodnotách do 350 MHz a každá buňka potřebuje na svůj výpočet deset šestivsupých LUTů.

```
Slice Logic Utilization:
 Number of Slice Registers:            1024  out of  106400     0%
 Number of Slice LUTs:                10240  out of  53200    19%
    Number used as Logic:             10240  out of  53200    19%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:  10240
   Number with an unused Flip Flop:    9216  out of  10240    90%
   Number with an unused LUT:             0  out of  10240     0%
   Number of fully used LUT-FF pairs:  1024  out of  10240    10%
   Number of unique control sets:      1024
```

5.3.Software

5.3.1. Xilinx ISE Design Suite 14.5

Xilinx ISE (**I**ntegrated **S**oftware **E**nviroment) je software pro syntézu a analýzu HDL (**H**ardware **D**escription **L**anguage) kódů. Vývojářům je zde umožněno kompilovat kód, simulovat své návrhy, zkoumat RTL (**R**egister **T**ransfer **L**evel) diagramy atd.

Jedná se o ověřený nástroj pro všechna programovatelná zařízení (FPGA, SoC).

5.3.2. Xilinx Platform Studio EDK

EDK (**E**mbedded **D**evelopment **K**it) je integrované vývojové prostředí pro návrh vestavěných systémů. Hlavní součástí tohoto setu je XPS (**X**ilinx **P**latform **S**tudio), který umožňuje „rozběhnutí“ hardwaru již během několika málo minut. XPS zahrnuje také inteligentní průvodce, umožňující rychlé nastavení architektury vestavěného systému, sběrnic a vstupně/výstupních periférií.

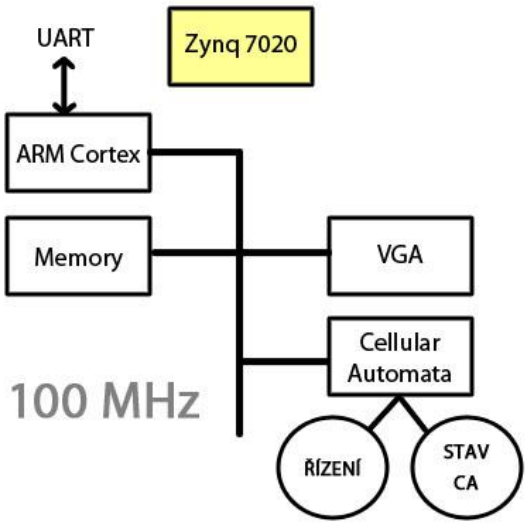
5.4.Testování na hardwaru

Vedoucím práce byl vytvořen kód, jenž umožnil demonstrovat generovaný VHDL návrh na reálném hardwaru – konkrétně na vývojové desce ZedBoard osazené AP SoC ZYNQ-7020 [1.5.1].

Celý výpočet automatu je řízen procesorem ARM Cortex komunikující se stolním počítačem přes UART[3]. Na obrázku [Obr. 34] je jednoduchá vizualizace principu zpracování celulárních automatů na AP SoC Zynq-7020 [33]. Přes UART se

[3] **U**niversal **A**synchronous **R**eceiver/**T**ransmitter – sériová/paralelní komunikace

načtou vstupní buňky, které se zpracují v bloku obsahujícím popis chování celulárních automatů a poté se výstup zobrazí pomocí VGA[4] rozhraní na připojené obrazovce.



Obr. 34 - Celulární automat na ZYNQ-7020

Výpočet jednoho kroku v celulárním automatu o rozměru 32 × 32 prvků (včetně nahrání vstupu a zobrazení výstupu) trvá přibližně 11 – 12 ms a v případě celulárního automatu 1024 × 768 trvá cca 168 ms. V porovnání s desktopovou aplikací je tento výpočet několikanásobně rychlejší – kroky v celulárním automatu o rozměru 32 × 32 se zobrazují v intervalu cca 30 ms a u CA 1024 × 768 se zpoždění počítá již na sekundy (cca 10 s).

Větší rozměr celulárních automatů se řeší rozložením plochy na okénka menších rozměrů, která se kvůli okrajovým podmínkám musejí překrývat. Výsledný návrh za použití okének o rozměru 32 × 32 pokryl 25% FPGA. Za použití větších okének o rozměru 64 × 64 bylo využití vyšší – 80% FPGA.

[4] **V**ideo **G**raphics **A**rray – počítačový zobrazovací (grafický) standard

6. Závěr

V rámci práce byl vyvinut program, který úspěšně zobrazuje všechny tři zamýšlené typy celulárních automatů – dva obecné (1D, 2D) a jeden konkrétní s pevně stanoveným pravidlem (Conwayovu hru života). Všechny body zadání se mi s pomocí mého vedoucího bakalářské práce podařily splnit a jsou v práci obsaženy.

Vývoj desktopové GUI aplikace začal v jazyce Java, protože to byl do té doby mnou nejlépe ovládaný jazyk. Později jsem se naučila s programovacím jazykem C#. Práce s tímto jazykem v prostředí MS Visual Studio 2010 se mi natolik zalíbila, že jsem se rozhodla pro experiment – předělat aplikaci z jazyka Java do C#. Pokus se vydařil a program v C# nahradil předchozí aplikaci psanou v Javě.

Algoritmus je poměrně jednoduchý, nevyužívá složitých funkcionalit. V podstatě se podle vektoru, který je tvořen zkoumaným prvkem a jeho okolím, vyhledá výsledný stav – vektor označuje index pole pravidla.

I přesto, že jsem během této práce nepoužívala složité algoritmy a nevyužívala vícero funkcí, které Java či C# nabízí, byla jsem obohacena o velkou zkušenost samostatného vývoje projektu a nadchlo mě to. Ráda bych se ve svých znalostech i nadále zdokonalovala, abych v budoucnu mohla podobnou činnost vykonávat jako zaměstnání na plný úvazek.

S programováním v jazyce VHDL mi velice pomohl můj vedoucí práce Ing. Martin Rozkovec, Ph.D., za což mu velmi děkuji. Během vývoje se několikrát měnil návrh realizace. Nakonec se došlo k závěru, že nejlepším řešením bude „jít“ buňku po buňce, přičemž každá buňka byla popsána jako samostatná komponenta a následně zahrnuta do pole celulárního automatu, kde se na základě pravidel vyčítal jejich následující stav.

Závěrem je zjištění, že výpočet jednotlivých kroků pomocí FPGA je mnohem rychlejší oproti simulaci na desktopové aplikaci vyvinuté v programovacím jazyce C# - při větších rozměrech sahá rozdíl až do řádů sekund.

Seznam použité literatury

[1] BALOUGH, CH. *Strategic considerations for emerging SoC FPGAs* [online]. Altera. San Jose, 02. 06. 2011 [cit. 2013-05-16]. URL: <http://www.techdesignforums.com/practice/technique/strategic-considerations-for-emerging-soc-fpgas/>.

[2] BĚHÁLEK, M. *Programovací jazyk C#* [online]. Katedra informatiky, VŠB-TU Ostrava. Ostrava, 2007 [cit. 2013-05-16]. URL: <http://www.cs.vsb.cz/behalek/vyuka/pcsharp/text/>.

[3] BERTO, F. – TAGLIABUE, J. *Cellular Automata: The 256 Rules* [online]. Stanford University, Stanford Encyclopedia of Philosophy. Copyright © 2012 [cit. 2013-05-16]. URL: <http://plato.stanford.edu/entries/cellular-automata/supplement.html>.

[4] BLÁHOVÁ, E. *Zásobníkový automat* [online]. Pedagogická fakulta, Univerzita Karlova. Praha, 2005 [cit. 2013-05-16]. URL: <http://class.pedf.cuni.cz/jancarik/download/ZasAut.pdf>.

[5] *Cellular automaton*. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-05-16]. URL: <http://en.wikipedia.org/wiki/Cellular_automaton>.

[6] *Conway's Game of Life*. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-05-16]. URL: <http://en.wikipedia.org/wiki/Conway's_Game_of_Life>.

[7] COVENEY, P. – HIGHFIELD, R. *Mezi chaosem a řádem: Hranice komplexity: Hledání řádu v chaotickém světě*. Vydání první. Praha: Mladá Fronta, 2003. ISBN 80-204-0989-0.

[8] *CPLD a FPGA 1.díl - představení obvodů* [online]. 16. července 2008, [cit. 2013-05-16]. URL: <http://pandatron.cz/?481&cpld_a_fpga_1.dil_-_predstaveni_obvodu>.

[9] DOSTÁL, H. *Teorie konečných automatů, regulárních gramatik, jazyků a výrazů: Konečný automat* [online]. Fakulta informatiky a managementu, Univerzita Hradec Králové. Hradec Králové, Copyright © 2008, [cit. 2013-05-16]. URL: <http://iris.uhk.cz/tein/teorie/konecnyAutomat.html>.

[10] HEMINGWAY, B. *Finite state machines* [online]. Department of Computer Science and Engineering, University of Washington. 2003, [cit. 2013-05-16]. URL: <http://www.cs.washington.edu/education/courses/cse370/03sp/pdfs/lectures/Lecture21.pdf>.

[11] HORDĚJČUK, V. *Turingův stroj* [online]. [cit. 2013-05-16]. URL: <http://voho.cz/wiki/matematika/turinguv-stroj/>.

[12] HOŘEJŠÍ, M. *C#: objektově orientovaný programovací jazyk* [online]. Fakulta informatiky, Masarykova univerzita. Brno, 2006 [cit. 2013-05-16]. URL: <http://www.fi.muni.cz/usr/jkucera/pv109/2006/xhorejsi.htm>.

[13] *Hra života*. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-05-16]. URL: <http://cs.wikipedia.org/wiki/Hra_%C5%BEivota>.

[14] HUSÁKOVÁ, M. *Celulární automaty: Znalostní technologie III materiál pro podporu studia* [online]. Univerzita Hradec Králové. Hradec Králové, 2007, č. 1, s. 14 [cit. 2013-05-16]. URL: <http://lide.uhk.cz/fim/ucitel/fshusam2/lekarnicky/zt3/zt3_dokumenty/CelularniAutomaty.pdf>.

[15] *Java (programovací jazyk)*. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-05-16]. URL: <http://cs.wikipedia.org/wiki/Java_(programovac%C3%AD_jazyk)>.

[16] JELEMENSKÁ, K. ed. *Počítačové architektúry a diagnostika: česko-slovenský seminár pre študentov doktorandského štúdia*. Bratislava: Slovenská technická univerzita v Bratislave, 2011. ISBN 978-80-227-3552-0.

[17] KOCUR, P. *Úvod do teorie konečných automatů a formálních jazyků* [online]. Plzeň, prosinec 2000, [cit. 2013-05-16]. URL: <http://www.kvd.zcu.cz/cz/materialy/kafj/_kafj1.html#_Toc501115421>.

[18] KOLOUCH, J. *Stavové automaty* [online]. Fakulta elektrotechniky a komunikačních technologií, VUT. [cit. 2013-05-16]. URL: <http://www.urel.feec.vutbr.cz/~kolouch/pld/1_prednasky/kapitola06.html>.

[19] KOLOUCH, J. *Stavové automaty* [online]. Fakulta elektrotechniky a komunikačních technologií, VUT. [cit. 2013-05-16]. URL: <http://www.urel.feec.vutbr.cz/~kolouch/pld/2_cviceni/kapitola05_05.html>.

[20] KOLOUCH, J. *Základní struktura modelu v jazyku VHDL* [online]. Fakulta elektrotechniky a komunikačních technologií, VUT. [cit. 2013-05-16]. URL: <http://www.urel.feec.vutbr.cz/~kolouch/pld/2_cviceni/kapitola03_02.html>.

[21] KOT, M. *Turingův stroj* [online]. Fakulta elektrotechniky a informatiky, Technická univerzita Ostrava. Ostrava, 2010, [cit. 2013-05-16]. URL: <http://www.cs.vsb.cz/kot/soubory_animaci/a-turinguv_stroj.pdf>.

[22] *MSDN – the Microsoft Developer Network* [online]. Microsoft, 2012 [cit. 2013-05-16. URL: <http://msdn.microsoft.com/en-US/>.

[23] PADRTA, D. *Základy programovacího jazyka Java* [online]. Copyright © 2007, [cit. 2013-05-16]. URL: <http://skola.isd.cz/java/03_zaklady_javy.pdf>.

[24] Panda, S. P. ed. *Encryption and Decryption algorithm using two dimensional cellular automata rules in Cryptography* [online]. 2011, issue-1, p. 6 [cit. 2013-05-16]. URL: <http://www.interscience.in/IJCNS_Vol1_No1/Paper_4.pdf>.

[25] PECH, J. *Nebojte se FPGA* [online]. 06. 02. 2006, [cit. 2013-05-16]. URL: <http://www.hw.cz/teorie-a-praxe/dokumentace/nebojte-se-fpga.html>.

[26] PELÁNEK, R. *Buněčné automaty* [online]. Fakulta informatiky, Masarykova univerzita. Brno, 2012 [cit. 2013-05-16]. URL: <http://www.fi.muni.cz/~xpelanek/IV109/slidy/ca.pdf>.

[27] PELIKÁN, J. *Základy jazyka Java (pro programátory v jazyce C++)* [online]. Matematicko-fyzikální fakulta, Skupina počítačové grafiky, Univerzita Karlova. Copyright © 2003-2004 [cit. 2013-05-16]. URL: <http://cgg.mff.cuni.cz/~pepca/java/JavaForC++.pdf>.

[28] *Popis jazyku VHDL* [online]. Fakulta elektrotechnická, ČVUT. Praha, Copyright © 2008-2013 [cit. 2013-05-16]. URL: <http://measure.feld.cvut.cz/cs/system/files/files/cs/vyuka/predmety/A0B38APH/VHDL.pdf>.

[29] SLINTÁK, V. *Základní konstrukce ve VHDL* [online]. 23. 07. 2012, [cit. 2013-05-16]. URL: <http://uart.cz/633/zakladni-konstrukce-ve-vhdl/>.

[30] *Table of Cellular Automaton Properties: 1986* [online]. 2011, issue-1 p. 72 [cit. 2013-05-16]. URL: <http://www.stephenwolfram.com/pdf/Cellular-Automaton-Properties-Stephen-Wolfram-Article.pdf>.

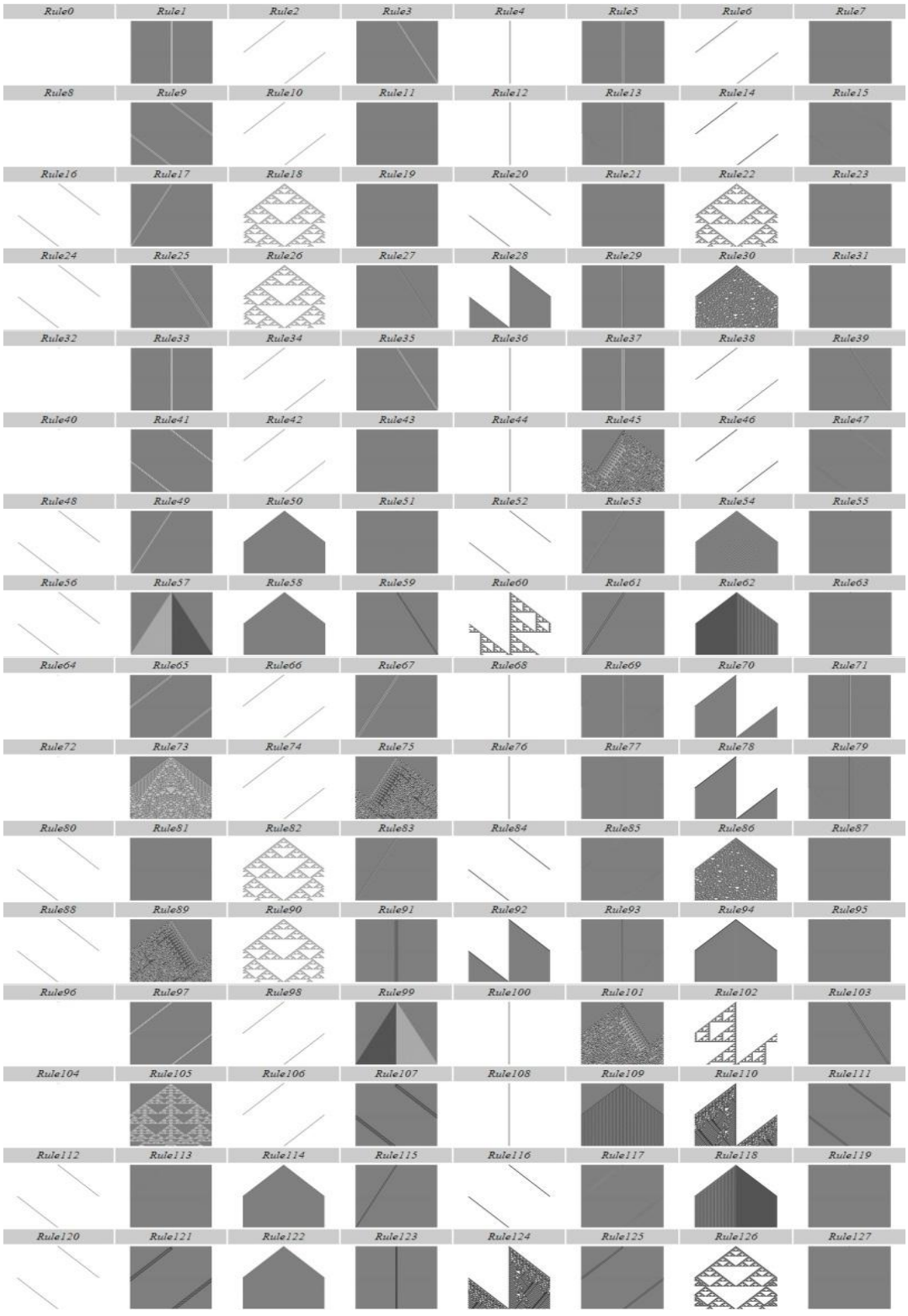
[31] TOMAŠTÍK, M. *Celulární automaty (Game of life)*. Brno, 2010. 37 s. Bakalářská práce. Fakulta strojního inženýrství, Vysoké učení technické v Brně. Vedoucí bakalářské práce: Ing. Radomil Matoušek, Ph.D.

[32] *VHDL*. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-05-16]. URL: <http://cs.wikipedia.org/wiki/VHDL>.

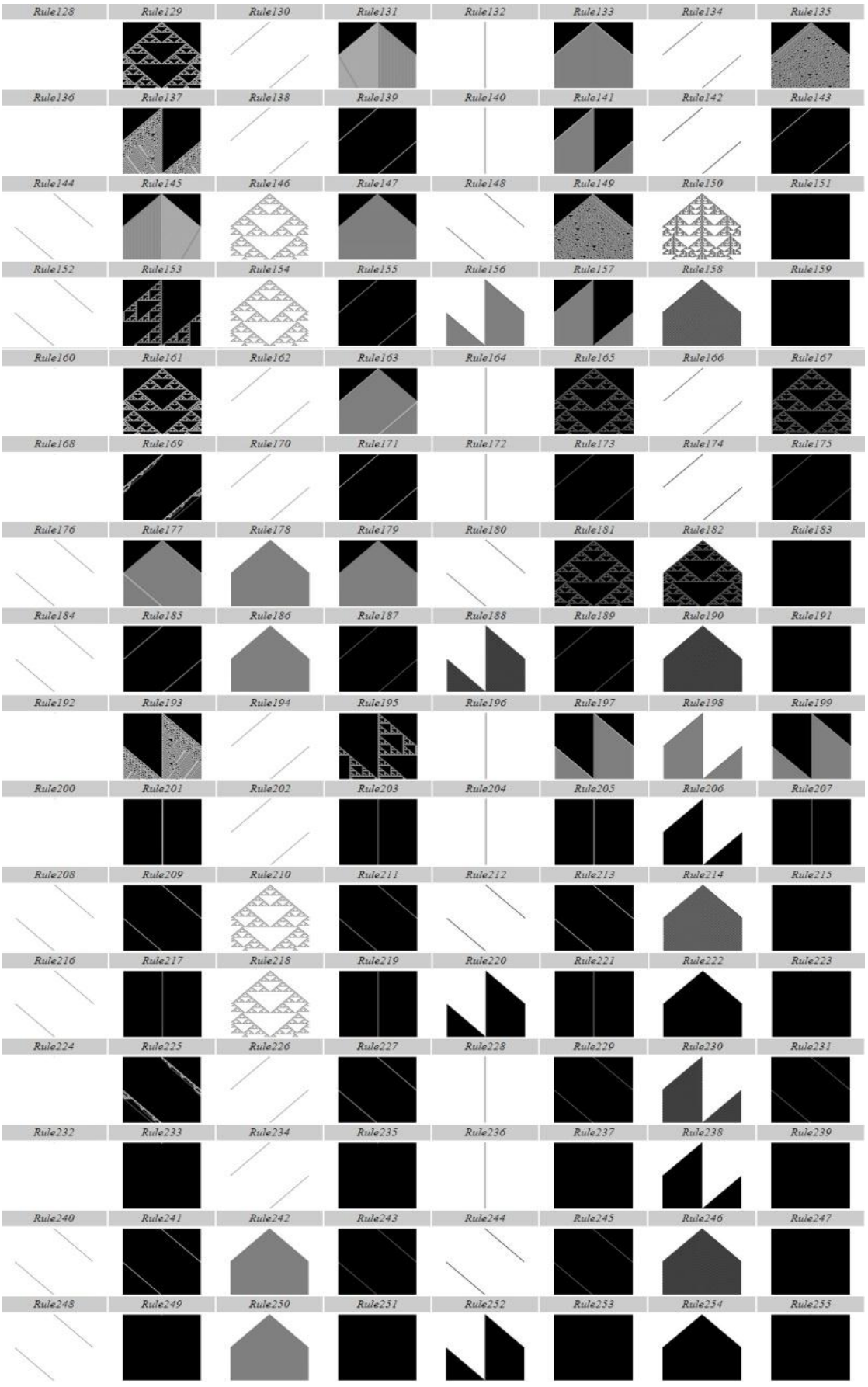
[33] *Zynq-7000 All Programmable SoC* [online]. © Copyright 2013 Xilinx Inc., [cit. 2013-05-16]. URL: <http://www.xilinx.com/content/xilinx/en/products/silicon-devices/soc/zynq-7000.html>.

A. Příloha

A.1. Vzory jednotlivých 1D pravidel

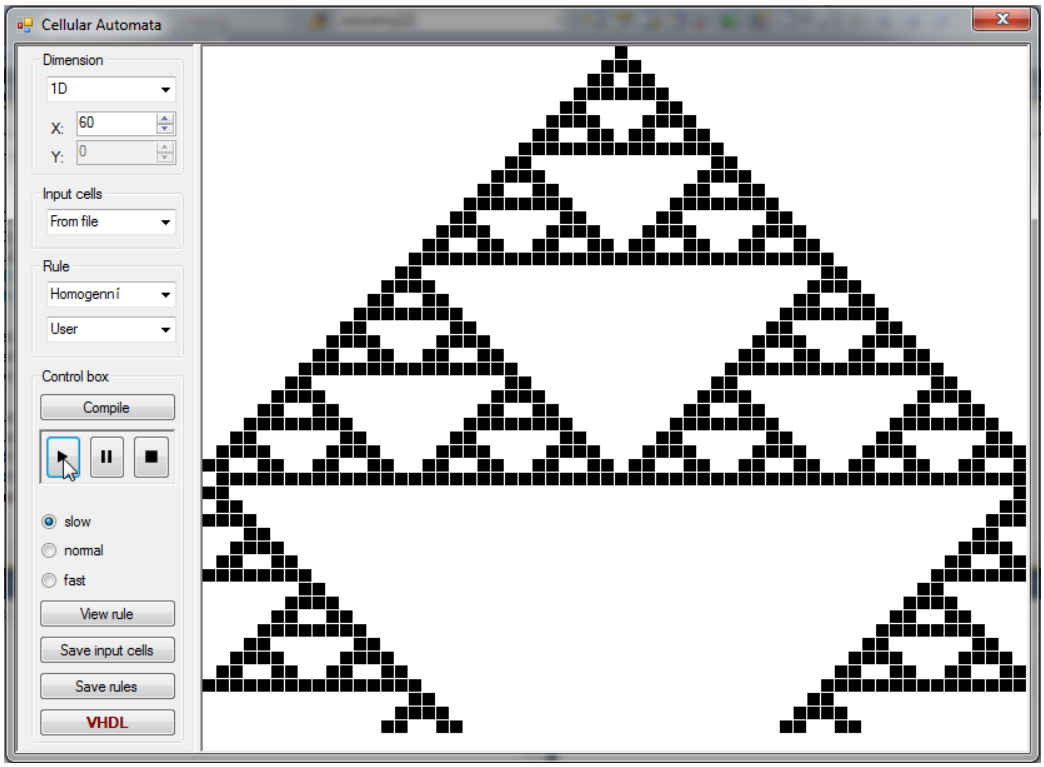


Obr. 35 - 1D CA s jednou vstupní živou buňkou (pravidlo 0 - 127) © [3]

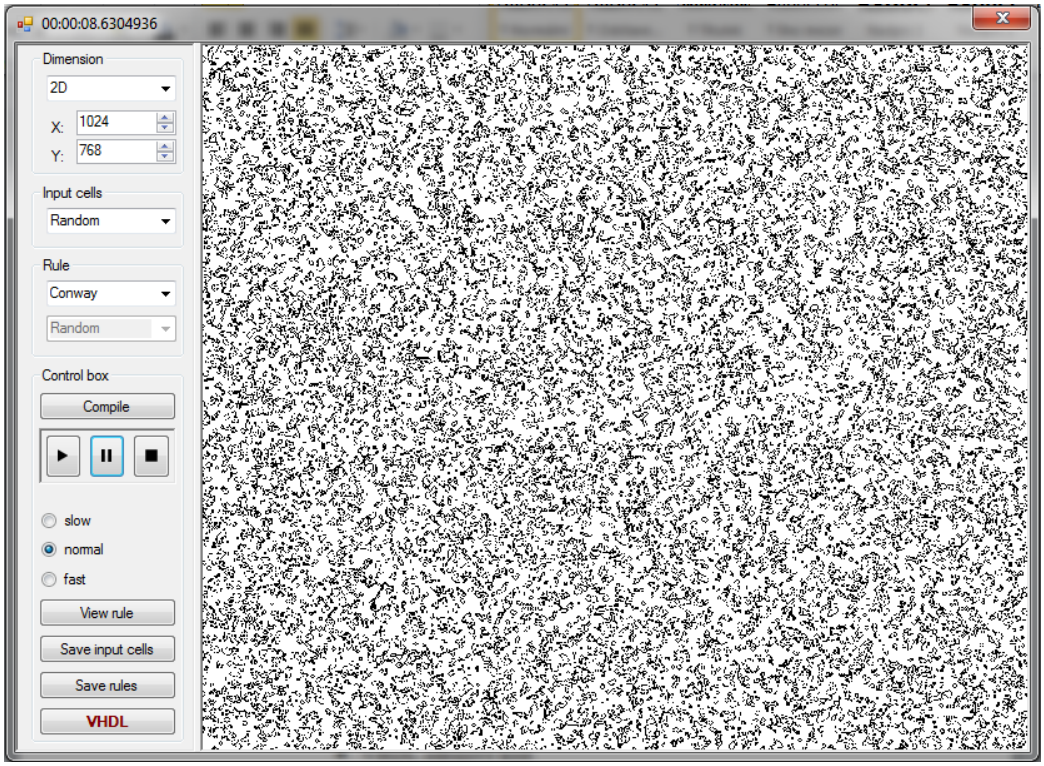


Obr. 36 - 1D CA s jednou vstupní živou buňkou (pravidlo 128 - 255) © [3]

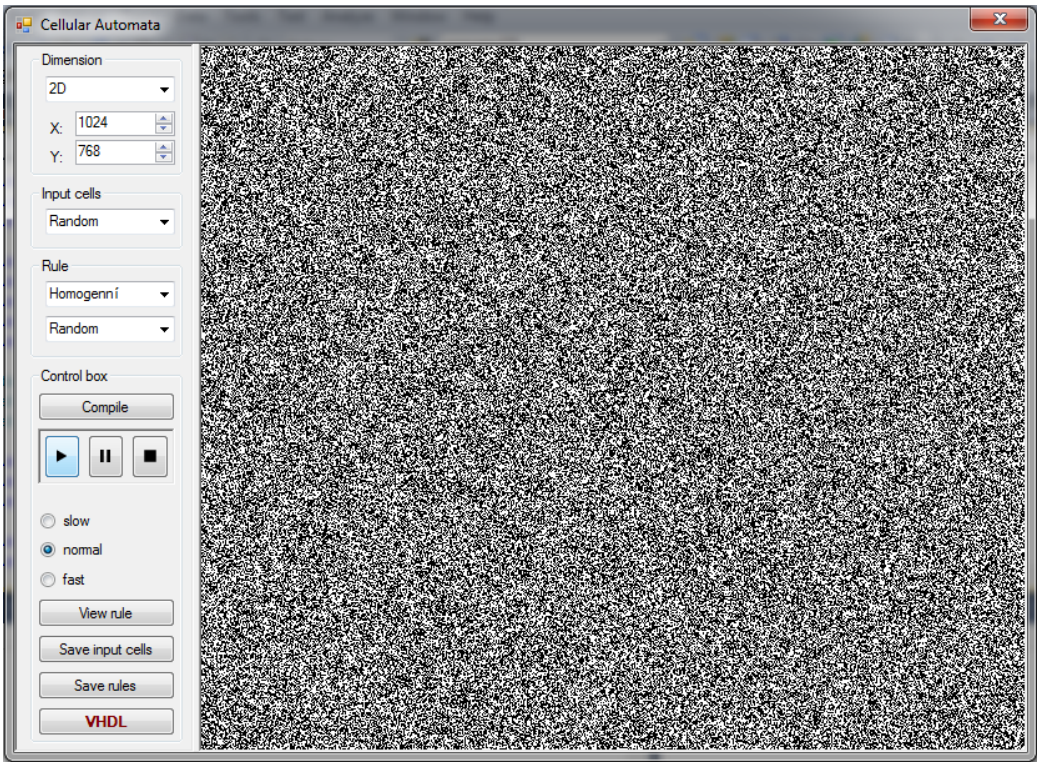
A.2. Zobrazení simulátoru CA



Obr. 37 - 1D CA pravidlo 126 (velikost buňky 10)



Obr. 38 - 2D CA Conwayova hra života (velikost buňky 1)



Obr. 39 - 2D CA náhodné homogenní pravidlo (velikost buňky 1)

A.3. Obsah adresářů na přiloženém CD

- Bakalářská práce
  - text práce ve formátu *.pdf
  - fulinova_veronika_bakalarska_prace_2012_2013.pdf
- Přílohy
  - použité obrázky
- Zdrojový kód
  - C# projekt psaný v MS Visual Studiu 2010
  - VHDL zdrojový kód